Windows环境下汇编语言程序设计

蓝琚成　肖金秀　编著

地质出版社

◎ 责任编辑／何 曼 张桂华
◎ 封面设计／凌 波

Windows 环境下汇编语言程序设计

ISBN 7-116-03383-1

9 787116 033832 >

ISBN 7-116-03383-1
T·88 定价：45.00 元（含磁盘）

Windows 环境下汇编语言程序设计

蓝瑶成　肖金秀　编著

地质出版社

· 北京 ·

内容简介

本书以汇编语言的形式介绍32位Windows环境下API编程和Windows内部操作机制。它从一个汇编程序员的角度，以丰富的实例详细解释了32位Windows的编程。它不仅告诉你"要这样做"，还告诉你"为什么要这样做"，使读者对Windows的消息循环、窗口处理过程、Windows中的分段和分页保护、中断等内部操作机制有一个深入的了解。

因为本书的程序是用汇编语言编写，这样使读者对程序及相应Windows操作的每一个步骤、每一个细节都有更清楚和更直观的了解。

本书从实用性、易掌握性出发，重点突出、操作简练、内容丰富，可操作性强。本书可作为高级编程人员的参考书，也可作为大专院校计算机专业的参考书。

图书在版编目（CIP）数据

Windows环境下汇编语言程序设计 / 蓝琚成等编著.
北京：地质出版社，2001.3
ISBN 7-116-03383-1

I. W… II. 蓝… III. 汇编语言－程序设计
IV. TP313

中国版本图书馆CIP数据核字（2001）第07769号

责任编辑：何蔓 张柱丰
出版发行：地质出版社
社 址：北京海淀区学院路29号，100083
电 话：010－82310758；82324580
网 址：http://www.gph.com.cn
传 真：010－82310759
印 刷：广东出版技校彩印厂印刷
开 本：787×1092 1/16
印 张：22.5
字 数：519千字
印 数：1-1200
版 次：2001年3月北京第一版・第一次印刷
定 价：45.00元（含磁盘）

ISBN 7-116-03383-1/T・88

（凡购买地质出版社的图书，如有缺页、倒页、脱页者，本社发行处负责调换）

前 言

目前，市面上关于 Windows 编程的书很多，所涉及的工具有 Visual Basic、Visual C++、Delphi、C++ Builder，有专门讨论图形编程的，有专门讨论 Windows Socket 编程的，如此等等。但是，很少会涉及如何使用汇编语言编写 Windows 程序。本书正是尝试从一个汇编程序员的角度来解释 Windows 编程。

在 DOS 的时代，程序员为了获得高性能或直接控制硬件而编写汇编程序。进入 Windows 时代，C 程序员都知道直接用 C 语言写 Windows 程序是如何麻烦的一件事——一个最简单的程序框架都要写上数十行的代码。于是，程序员自然不会想到用汇编语言来写 Windows 程序。再加上 Windows 系统本身的保护机制，对程序员直接控制硬件造成困难，因此，喜欢用汇编语言编写 Windows 程序的程序员并不多。但是，在这里要说的是，用汇编语言编写 Windows 程序并不会比用 C 语言困难。现在的汇编编译器如 Tasm、Masm 都提供对宏、结构的支持，还加入了对面向对象思想的支持，这些都大大降低了我们用汇编语言编写 Windows 程序的难度。

现在，为了简化程序员的编程工作、提高工作效率，各种 IDE 工具如 Visual C++和 Delphi 的编译环境及其伴随的类库 MFC 和 OWL 对 Windows API 的封装，大大方便了程序员的工作。这些集成环境和类库使我们不用重复编写大量类似的代码，并增强程序的可读性、可维护性等，因此，使用集成开发环境确实是一个不错的选择。但是，不少人仍建议那些学习 Windows 编程的人尝试直接使用 C 语言和 SDK 来编程，因为只有这样，才能最直观地了解 Windows 的运作机制，也只有这样，当使用 MFC 和 OWL，才容易明白那些类为什么是这样而不是那样封装，消息循环到底是怎么一回事等等。我们并不赞成您在编写实用的 Windows 程序时大量使用汇编语言，正如你不应该直接用 C 语言和 API 而应该使用 MFC 或 OWL 写一个大型程序一样，除非你要写的是一个设备驱动程序之类的东西。但是，使用过汇编语言编写程序后，您不但可以得到尝试用 C 语言和 SDK 写程序的好处，还可以对数据及代码的存储形式、函数调用过程中的参数传递、堆栈框架的创建和注销过程以及系统的内部运作过程有一个更深入的了解。

本书从基础讲起，第 1、2 章介绍 80X86 系列 CPU 的汇编语言编程。汇编语言的基础知识是深入学习 Windows 下汇编编程的必备条件。事实上，对于 Windows 的一些内部机制，若没有汇编语言的知识是不可能理解的。

作为 Windows 汇编编程的入门，第 3 章讲述 Windows 汇编编程的基本步骤，介绍普通 Windows 程序、动态链接库（DLL）、控制台应用程序及虚拟设备驱动程序（VxD）的编写方法、编译步骤，以及与 Windows API 相关的一些基础知识。

内存管理、进程管理、I/O 管理和文件管理是操作系统的 4 个基本功能。第 4、5、6、7 章分别从这 4 个方面介绍 Windows 的内存模式、进程和线程的创建及调度、中断和 I/O 机制、文件系统，以及它们的编程。

第 8 章着重介绍如何用汇编语言编写 Windows 的图形及界面程序。

Windows 的 PE 文件格式是 Windows 可在各种机器上实现的基础之一，第 9 章讲述 PE 文件方面的知识。

在本书后面的附录中，我们给出编译本书程序的编译器的设置方法；给出用另一种编译器Masm的使用方法；另外列出流行的奔腾和MMX指令的说明。

本书中的DOS程序用MASM 5.0或Tasm 5.0都可以编译通过，Windows程序用Tasm 5.0或Visual C++ 6.0编译，在Windows 98和Windows 2000上调试。这些Windows程序要移植到Masm或C++ Builder的环境中编译相信不会太困难。

我们虽然很希望书中的内容没有错误和程序没有bug，但是，我们知道这实际上是不可能的，欢迎您对本书的内容或程序提出意见或建议，可以给我们发E-mail：einar@163.net和thzc@21cn.com。我们期待在Windows和汇编语言的世界里认识您。

最后，我们要感谢杨文龙老师，这本书的出版离不开他的帮助和指导。

作　者

2001年2月于广州

目 录

第一章 80X86 的寄存器及指令系统 1

1.1 80386 处理器概述 1

1.1.1 寄存器 1

1.1.2 寄存器使用实例 4

1.2 指令系统 11

1.2.1 寻址方式 11

1.2.2 指令系统 11

1.2.3 数据传送指令 11

第二章 保护模式程序设计 16

2.1 保护模式概述 16

2.2 进入保护模式 18

2.3 任务转换 21

2.4 分页内存管理 27

2.4.1 分页的功能 27

2.4.2 页保护 29

2.4.3 分页实例 30

2.5 虚拟 86 方式 35

第三章 用汇编语言编写 Win32 应用程序 44

3.1 在 C/C++中插入汇编指令 44

3.2 编写 Win32 汇编程序 45

3.2.1 Win32 汇编编程入门实例 45

3.2.2 中级编程实例 47

3.2.3 编写动态链接库 66

3.2.4 编写控制台应用程序 68

3.3 编写虚拟设备驱动程序 71

3.3.1 VxD 概述 71

3.3.2 动态 VxD 71

3.3.3 静态 VxD 81

第四章 Windows 95/98/NT 内存管理 84

4.1 Windows 95/NT 内存布局 84

4.2 操纵内存 88

4.2.1 虚拟内存 88

4.2.2 堆管理 91

4.2.3 内存映射文件 95

4.3 写时拷贝 109

目 录

4.4 在 Windows 95 里读写物理内存 114

第五章 进程和线程 124

5.1 Win32 进程 124
5.2 WinMain 函数的入口 126
5.3 建立线程和进程 128
5.4 线程调度和使用 131
5.5 写入远程进程 156
5.6 钩子过程 171

第六章 中断与 I/O 操作 179

6.1 80X86 的中断与异常 179
6.2 结构化异常处理 186
6.3 硬件中断的映射 190
6.4 给软中断挂钩 198
6.5 虚拟化 IRQ 204
6.6 I/O 操作 214
6.6.1 I/O 操作概述 214
6.6.2 Windows 9X 的 I/O 操作 215

第七章 文件系统 223

7.1 Windows 9X 的 FAT 和长文件名 223
7.2 文件操作 226
7.2.1 建立文件 226
7.2.2 定位文件指针函数 227
7.2.3 写文件 227
7.2.4 读文件 227
7.2.5 关闭文件 228
7.2.6 例程 228
7.3 Windows NT 下绝对磁盘的读写 230
7.4 Windows 9X 下绝对磁盘的读写 236
7.4.1 在 Win32 程序中调用 DOS 功能 236
7.4.2 DOS 功能调用 Int 21H 的 7305H 功能 236
7.4.3 使用 DeviceIoControl 的 VWIN32_DIOC_DOS_DRIVEINFO 功能 237
7.4.4 例程 238
7.4.5 Windows 95 中读写物理磁盘 242

第八章 图形及界面编程 247

8.1 图形接口 API 247

8.2 菜单........253
8.3 子窗口........262
8.4 DirectX........267
8.4.1 COM 对象在内存中的表示........268
8.4.2 DirectDraw........270
8.5 遍历窗口........280

第九章 PE 文件格式........289

9.1 PE 文件结构........289
9.1.1 DOS 插桩程序........289
9.1.2 NT 映像头........293
9.1.3 节表........299
9.1.4 节........301
9.2 PE 文件查看器........308
9.2.1 Tasm 的文件查看器........308
9.2.2 Windows “快速查看”........310
9.2.3 PEDump 文件查看器........312

附 录........316

A.1 系统环境设置........316
A.2 用 Masm 编写 Windows 程序........317
A.3 Pentium 汇编指令........322
A.4 MMX 指令和寄存器........337

第一章 80X86的寄存器及指令系统

本章主要介绍80X86的寄存器及指令系统。即寄存器及使用、寻址方式、指令系统及数据传送指令等内容。

1.1 80386处理器概述

1.1.1 寄存器

1. 通用寄存器

自80386开始，寄存器的大小和个数发生了变化，16位通用寄存器ax、bx、cx、dx、si、di、bp、sp都扩展成32位，分别记为eax、ebx、ecx、edx、esi、edi、ebp、esp。指令指针寄存器ip亦变成32，记为eip，如图1-1所示。

31　　16	15　　0
EAX	AX
EBX	BX
ECX	CX
EDX	DX
ESI	SI
EDI	DI
EBP	BP
ESP	SP
EIP	IP

图1-1

扩展后的32位寄存器的低16位仍可如原来一样使用。如：

```
mov eax,12345678H
mov ax,1234H
mov al,12H
```

这样做的理由是为软件的兼容性打下基础。

指令指针寄存器虽然扩展成32位，但并非总是使用32位。只有在执行保护模式下的32位段代码时才使用eip的高16位，而在执行实模式或保护模式的16段代码时，只使用低16位的ip。

2. 段寄存器

8086的段寄存器cs、ds、es、ss是16位，在80386中没有变动，但增加了2个段寄存器：fs和gs，它的位的用法与es类似，都是扩展数据段寄存器，如：

```
mov eax,fs:[ebx]
```

但段寄存器在保护模式下的意义与实模式不相同，这在后面有讲述。

3. 标志寄存器

8086 的标志寄存器是 16 位。在 80386 及其后续兼容 CPU 中，为了支持任务切换、IO 端口保护、虚拟 8086 方式、分页技术等，增加了一些标志位，并将标志寄存器扩展成 32 位，记作 eflags，如图 1-2 所示。

31	17	16	15	14	13 12	11	10	9	8	7	6	5	4	3	2	1	0
	VM	RF	0	NT	IOPL	OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

图 1-2

它的位为 0，与 8086 相同的位我们不再细述，只讲述新增的位：

1）VM（第 17 位）：虚拟 86 方式标志，当此位为 1 时，CPU 在虚拟 86 方式下工作。这时 CPU 还是在保护模式下，但其指令执行类似于实模式。虚拟 86 方式的切入切出是很复杂的，并非只要设置此位即可。

2）RF（第 16 位）：Restart 标志，控制是否接受调试故障。RF=0 为接受，否则不接受。CPU 每完成一条指令都会把 RF 填 0。当接收到一个非调试故障时，把 RF 填 1。

3）NT（第 14 位）：任务嵌套标志，当此位为 1 时，指令 iret 实现任务转换，否则中断返回。

4）IOPL（第 12、13 位）：IO 保护标志，当当前特权级的值小于等于此 IOPL 的值时，可执行 IO 指令。

4. 控制寄存器

80386 有三个控制寄存器：CR0、CR2、CR3，用来控制机器的全局状态，CR1 未定义，如图 1-3 所示。

	31 … 16	15 …	4	3	2	1	0
CR0	PG		ET	TS	EM	MP	PE
CR2							
CR3							

图 1-3

其中 CR0 的低 16 位也称为机器状态字 MSW，这是 80286 遗留下来的概念。

CR0 各位的意义如下：

1）PE（Protection Enable）：保护模式允许位。当 PE 位为 1 时处于保护模式下（但不能通过简单地置位来切入保护模式），系统启动时此位置 0。

2）TS（Task Switched）：任务切换位。当发生任务切换时，此位置为 1。

3）MP（Monitor Coprocessor）：协处理器监控位。当任务切换，TS 置 1 时，如 MP 为 1，则执行 wait 指令时会产生一个“协处理器无效”异常。

4）EM（Emulate Coprocessor）：模拟协处理器。当此位为 1 时，协处理器指令会产生一个“协处理器无效”异常。这时可用软件来模拟协处理器。

5）ET（Processor Extension Type）：处理器扩展类型。若此位置为 1，则使用 32 位数据类型，即协处理器是 80387 或以上。

6）PG（Pageing Enable）：分页功能。若此位为 1，则正处于分页模式下（同样，不

能通过简单地置位来使其能分页）。注意 PG 为 1 时 PE 也必须为 1。

CR2 和 CR3 是在分页管理时用到。

1）CR2：页故障地址寄存器。当发生页故障时，CR2 存放有故障页的线性地址。

2）CR3：是页表目录寄存器。用作存放页表目录的物理基地址。

这些寄存器的具体用法将在后面的章节中讲述。

另外，80486/Pentium 在控制寄存器的数目和控制位上都有了扩展，这里不详述。

5. 调试、测试寄存器

- 调试寄存器

调试寄存器是 DR0~DR3、DR6、DR7，如图 1-4 所示。

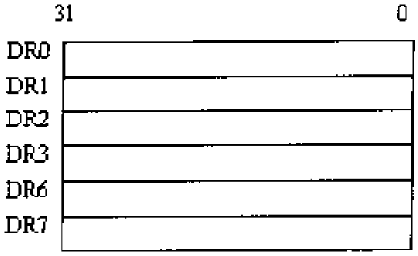


图 1-4

1）DR0~DR3 用作设置断点。

2）DR6 用作保存断点状态。

3）DR7 用作控制断点。

- 测试寄存器

TR6~TR7 是测试寄存器，用作存储器测试，如图 1-5 所示，它们都是 32 位寄存器。

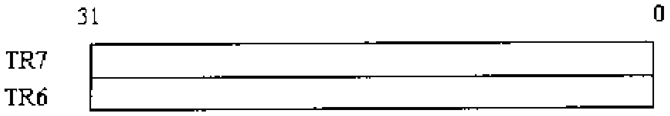


图 1-5

6. 系统地址寄存器

1）GDTR 称为全局描述符表。存放全局描述符的线性地址和段界限。

2）IDTR 称为中断描述符表。存放中断描述符的线性地址和段界限。

3）TR 任务状态段的选择子。

4）LDTR 存放局部描述符 LDT 的选择子。

这 4 个寄存器如图 1-6 所示。

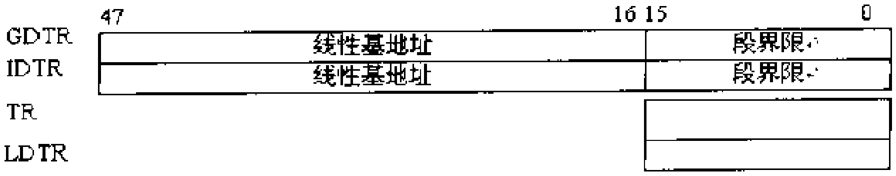


图 1-6

这些寄存器的具体用法我们在后面的章节中讲述。

1.1.2 寄存器使用实例

1. 程序代码

```
;****************************
;文件：ShowReg.asm          *
;功能：显示 80X86 的寄存器  *
;****************************
.386P
Data Segment use16
Begin    label byte
    _CS     db 'CS =     ',0DH,0AH
    _IP     db 'IP =     ',09H
    _DS     db 'DS =     ',0DH,0AH
    _ES     db 'ES =     ',09H
    _FS     db 'FS =     ',0DH,0AH
    _GS     db 'GS =     ',09H
    _SS     db 'SS =     ',0DH,0AH
    _ESP    db 'ESP=     ',09H
    _EAX    db 'EAX=     ',0DH,0AH
    _EBX    db 'EBX=     ',09H
    _ECX    db 'ECX=     ',0DH,0AH
    _EDX    db 'EDX=     ',09H
    _ESI    db 'ESI=     ',0DH,0AH
    _EDI    db 'EDI=     ',09H
    _EBP    db 'EBP=     ',0DH,0AH

    _CR0    db 'CR0=     ',09H
    _CR2    db 'CR2=     ',0DH,0AH
    _CR3    db 'CR3=     ',09H
    _DR0    db 'DR0=     ',0DH,0AH
    _DR1    db 'DR1=     ',09H
    _DR2    db 'DR2=     ',0DH,0AH
    _DR3    db 'DR3=     ',09H
    _DR6    db 'DR6=     ',0DH,0AH
    _DR7    db 'DR7=     ',09H
    _EFL    db 'EFL=     ',0DH,0AH
```

```
    _LDTR  db 'LDTR=     ',0DH,0AH
    _GDTR  db 'GDTR=     ',0DH,0AH
    _IDTR  db 'IDTR=     ',09H
        db '$'
    Digit     db '0123456789ABCDEF'
Data Ends
;++++++++++++++++++++++++++++++
Code Segment use16
    assume cs:code,ds:data,es:data
Start:
    mov ax,Data
    mov ds,ax
    mov es,ax
    mov ax,cs
    lea di,[_CS+4]
    call Compute16
    mov ax,ds
    lea di,[_DS+4]
    call Compute16
    mov ax,es
    lea di,[_ES+4]
    call Compute16
    mov ax,fs
    lea di,[_FS+4]
    call Compute16
    mov ax,gs
    lea di,[_GS+4]
    call Compute16
    mov ax,ss
    lea di,[_SS+4]
    call Compute16
    call Next
Next:
    pop ax
    lea di,[_IP+4]
    call Compute16

    mov eax,eax
```

```
lea di,[_EAX+4]
call compute32
mov eax,ebx
lea di,[_EBX+4]
call compute32
mov eax,ecx
lea di,[_ECX+4]
call compute32
mov eax,edx
lea di,[_EDX+4]
call compute32
mov eax,esi
lea di,[_ESI+4]
call compute32
mov eax,edi
lea di,[_EDI+4]
call compute32
mov eax,ebp
lea di,[_EBP+4]
call compute32
mov eax,esp
lea di,[_ESP+4]
call compute32

mov eax,cr0
lea di,[_CR0+4]
call compute32
mov eax,cr2
lea di,[_CR2+4]
call compute32
mov eax,cr3
lea di,[_CR3+4]
call compute32

mov eax,dr0
lea di,[_DR0+4]
call compute32
mov eax,dr1
```

```
        lea di,[_DR1+4]
        call compute32
        mov eax,dr2
        lea di,[_DR2+4]
        call compute32
        mov eax,dr3
        lea di,[_DR3+4]
        call compute32
        mov eax,dr6
        lea di,[_DR6+4]
        call compute32
        mov eax,dr7
        lea di,[_DR7+4]
        call compute32

        pushfd
        pop eax
        lea di,[_EFL+4]
        call compute32

;       sldt ax
;       lea di,[_LDTR+5]
;       call Compute16

        push eax
        push ax
        sgdt [esp]
        pop ax
        lea di,[_GDTR+5]
        call Compute16
        add di,4
        pop eax
        call Compute32

        push eax
        push ax
        sidt [esp]
        pop ax
```

```
        lea di,[_IDTR+5]
        call Compute16
        add di,4
        pop eax
        call Compute32

        mov ah,9
        mov dx,offset Begin
        int 21H
        mov ah,4cH
        int 21H

compute32:
        pushad
        mov bx,offset Digit
        mov cx,8
        cld
again:
        rol eax,4
        push eax
        and eax,0Fh
        xlat
        stosb
        pop eax
        loop again
        popad
        ret
compute16:
        pushad
        mov bx,offset Digit
        mov cx,4
        cld
again1:
        rol ax,4
        push ax
        and ax,0Fh
        xlat
        stosb
```

```
        pop ax
        loop again1
        popad
        ret
Code Ends
        end Start
```

2. 编译与链接

用 Tasm 编译，然后用 Tlink 链接，如下：

```
E:\MYPROG\Asm32\Bk\Chapter1>tasm showreg.asm
E:\MYPROG\Asm32\Bk\Chapter1>tlink showreg.obj
```

3. 运行

在 Windows 98 的 DOS 窗口中程序运行结果如下：

```
E:\MYPROG\Asm32\Bk\Chapter1>showreg
CS =0C10
IP =003A            DS =0BF8
ES =0BF8            FS =0000
GS =0000            SS =0BF8

ESP=00000000        EAX=0000003A
EBX=00000000        ECX=000000FF
EDX=02FF0BE8        ESI=FFFF0000
EDI=0000008A        EBP=02DF0912
CR0=00000000        CR2=00000000
CR3=00000000        DR0=00000000
DR1=00000000        DR2=00000000
DR3=00000000        DR6=00000000
DR7=00000000        EFL=00003246

LDTR=
GDTR=0FFFC0F59000
IDTR=02FFC000D4A4
E:\MYPROG\Asm32\Bk\Chapter1>
```

在纯粹 DOS 环境下运行的结果略有不同。

下面是在载入 Himem.sys 和 Emm386.exe 的 DOS 下运行的结果：

```
CS =1226
IP =003A            DS =120E
ES =120E            FS =0000
OS =0000            SS =120E
```

ESP=00000000	EAX=0000003A
EBX=00000000	ECX=000000FF
EDX=001911FE	ESI=00450000
EDI=0FFF008A	EBP=00450912
CR0=00000011	CR2=00000000
CR3=00119000	DR0=00119000
DR1=00119000	DR2=00119000
DR3=00119000	DR6=00119000
DR7=00119000	EFL=00007246

LDTR=

GDTR=017700163890

IDTR=07FF00163090

下面是在没有载入 Himem.sys 和 Emm386.exe 的 DOS 下运行的结果：

CS =19B9

IP =003A	DS =19A1
ES =19A1	FS =0000
GS =0000	SS =19A1

ESP=00000000	EAX=0000003A
EBX=00000000	ECX=000000FF
EDX=00191991	ESI=00000000
EDI=0FFF008A	EBP=00000912
CR0=00000010	CR2=00000000
CR3=00000000	DR0=00000000
DR1=00000000	DR2=00000000
DR3=00000000	DR6=FFFF0FF0
DR7=00000400	EFL=00207246

LDTR=

GDTR=FFFF00086C14

IDTR=03FF00000000

为什么在 DOS 下也有差别呢？这是因为当起动 DOS 时，如果载入了 Emm386.exe，它会把 CPU 切换入保护模式，所以控制寄存器和系统地址寄存器会有不同。

另外要注意，上面 CRn 的值并不全是正确的，因为指令“mov crn,reg”是特权指令，只有在实模式或保护模式 0 环上执行的程序才可直接读出，否则会引起异常。

上面的 CRn 值除未载入 Emm386.exe 的 DOS 的那一次运行结果外，都是操作系统截获了这个异常后模拟这条指令产生的结果，而操作系统未必会把真实的值给我们。

例如，Windows 9X 的 CR0，CR2 和 CR3 都不可能为 0。

1.2 指令系统

1.2.1 寻址方式

与 8086 比较，80386 等 32 位处理器的寻址方式更灵活，这并不仅仅表现在可以用 32 位偏移量来寻址，例如：

```
mov   bx,[ax]
```

在 8086 中，这条指令是不允许的，但在 80386 及其后继 CPU 中是可以的。它还支持在变址寄存器上乘上一个比例因子 1、2、4 或 8，例如：

```
mov eax,[edi + esi*8]
```

其 32 偏移量可以如下：

EAX		EAX			
ECX		ECX			
EDX		EDX	1		
EBX	+	EBX *	2	+	8 位立即数
ESI		ESI	4		32 位立即数
EDI		EDI	8		
EBP		EBP			
ESP					

1.2.2 指令系统

80386 的指令系统保持了对以前的机器指令兼容，并且作了很大的扩展，使寻址更灵活，指令选择的余地更大，并增强了对高级语言的支持。下面介绍它的指令，主要介绍新增的部分，对在 8086 中原有的只略提一下。

另外，下面说明新增的指令只是相对于 8086 而言，有些指令在 80186、80286 中早已存在，并非都是 80386 指令系统新增的。

1.2.3 数据传送指令

1. 通用传送指令

通用传送指令主要是 mov、MOVZX、MVSX、xchg、push、pop、popa。

MOV 指令是以前的处理器就有的，MOVZX 和 MOVSX 是新增的。这两条指令支持不同长度的寄存器之间或寄存器与内存之间的数据传送，例如：

```
MOVZX   EAX,BL
MOVZX   DWORD PTR[MEM],AX
MOVSX   BX,AL
```

MOVSX WORD PTR[MEM],AL

MOVZX 按"无符号扩展"来传送，即扩展的高位置 0；MOVSX 则按"带符号扩展"来扩展，即原来最高位是什么，则扩展后的位就补什么。例如：

```
mov al,70H
movzx ax,al        ;ax 得 0070H
movsx ax,al        ;ax 得 0070H
mov   al,80H
movzx ax,al        ;ax 得 0080H
movsx ax,al        ;ax 得 FF80H
```

xchg 是数据交换指令，如：

```
xchg eax,ebx       ;交换 eax 和 ebx 的内容
```

push、pop、pusha、popa 是堆栈操作指令。其中 push 和 pop 除了增加对 32 位的扩展外，还增加了支持立即数的操作，例如：

```
push dword ptr 12345
```

指令 pusha 或 popa 是英语 push all 和 pop all 的缩写，它们一次过把所有通用寄存器压栈和出栈。

它有两种形式 pusha、pushad 及 popa、popad，分别是对 16 位和 32 位寄存器进行栈操作。

pushad 的入栈顺序是：EAX、ECX、EDX、EBX、ESP、EBP、ESI、EDI，pusha 与此类似。

2. 累加器传送指令

累加器的传送指令有 in、out、xlat 三条。

in 和 out 指令与 8086 的使用方法一样；xlat 则针对 32 寻址作了扩展，可以是 xlat 和 xlatb，其中 xlat 以 bx 为基地址，xlatb 是以 ebx 为基地址。

3. 标志传送指令

标志传送指令有以下四条：lahf、sahf、pushf、popf。

其中 pushf 和 popf 对 32 位标志寄存器作了扩展：pushfd、popfd。这两组指令分别对 16 位和 32 位的标志寄存器 flag 和 eflag 进行压栈和出栈操作。

4. 地址传送指令

地址传送指令有 5 个：lds、les、lfs、lgs、lss。它把存储器中的内容送入段寄存器和指定的通用寄存器。其中 lfs 和 lgs 是为新增的两个段寄存器而设的。

指令的使用如下：

```
lfs ebx,memory1
lgs edi,memory2
```

5. 数据类型转换指令

数据类型转换指令包括：cbw、cwd、cwde、cdq。它们的功能如下：

```
cbw     ; al 带符号扩展到 ax
cwd     ; ax 带符号扩展到 dx 和 ax
```

cwde ；ax 带符号扩展到 eax

cwq ；eax 带符号扩展到 edx 和 eax

6. 算术运算指令

80386 指令系统的算术运算指令与 8086 并没有很大的区别，最大的改变只是对 32 位寄存器的支持。例如：

add eax,ebx

7. 逻辑指令

逻辑指令包括 not、and、、or、xor、test、shl、shr、sal、sar、rol、ror、rcl、rcr、shrd、shld。前面的 13 个指令是 8086 中就有的，但它们增加了对 32 位寄存器的支持。移位指令还增加了一个特征，即对大于 1 的立即数的支持，例如要将 ax 左移 3 位，在 8086 中必须写成：

mov cl,3

shl ax,cl

但现在可以直接写成：

shl ax,3

最后两个指令是 80386 指令系统新增的 64 位移位指令，例如：

shld eax,ebx,10 ；eax 左移 10 位，低 10 位由 ebx 高 10 位补上，ebx 不变

shrd eax,ebx,10 ；eax 右移 10 位，高 10 位由 ebx 低 10 位补上，ebx 不变

需注意，这 2 条指令不改变第二个操作数。

8. 串操作指令

串操作指令包括 movs、cmps、scas、lods、stos、ins、outs。前 5 个指令是 8086 中原有的，80386 指令系统增加了对 32 寄存器的支持，如 movs 可以写成 movsb、movsw、movsd。它们的源变址寄存器、目的变址寄存器及计数器是 esi、edi、ecx 还是 si、di、cx，由它所在的段决定。若是 32 位段，则使用前者，否则使用后者。

新增的 ins 和 outs 指令可以是 insb、insw、insd 和 ousb、ousw、outsd，它对端口输入/输出连续的字串，变址寄存器是 esi，端口号要放在 edx 中。

9. 控制转移指令

控制转移指令包括如 jmp、jz、loop、call、ret、int、into、iret 等，它们与在 8086 中的意义基本相同。

但是，jz、jb 等条件跳转指令的跳转范围不再局限在−128~127 之间，而与 jmp 指令一样，不过，这时生成的指令长度要长一些。

80386 指令系统针对 jcxz 指令增加了一个指令 jecxz。它的跳转是以 ecx 是否为 0 而不是 cx 是为 0 作为标准。这个指令的跳转范围仍然是−128~127。

循环指令 loop 的跳转范围也仍然是−128~127。

10. 条件设置指令

80386 指令系统新增条件设置指令来支持编译程序和高级语言生成的代码。这些指令根据当前的 eflag 中的标志来设置对应的寄存器或存储器。例如下面的指令，当 eax 与 ebx 相等时设置 al 为 1，否则为 0：

cmp eax,ebx

setz al

条件设置指令如表 1-1 所示。

表 1-1 80386 指令系统新增条件设置指令的功能

指令	功能
Setz 或 sete	ZF=1 则置 1
Sets	SF=1 则置 1
Seto	OF=1 则置 1
Setp 或 setpe	PF=1 则置 1
Setb 或 setnae 或 setc	CF=1 则置 1
setbe 或 setna	CF=1 或 ZF=1 则置 1
Setl 或 setnge	SF 和 OF 不相等则置 1
Setle 或 setng	SF 和 OF 不相等或 ZF=1 则置 1
setnz 或 setne	ZF=0 则置 1
Setns	SF=0 则置 1
Setno	OF=1 则置 1
setnp 或 setpo	PF=0 则置 1
setnb 或 setae 或 setnc	CF=0 则置 1
setnbe 或 seta	CF=0 且 ZF=0 则置 1
Setnl 或 setge	SF 和 OF 相等则置 1
setnle 或 setg	SF 和 OF 相等且 ZF=0 则置 1

11. 标志指令

标志指令与 8086 一样，包括 clc、stc、cmc、cld、std、cli、sti。

12. 位处理指令

80386 指令系统新增位处理指令，它们的处理对象是位，而不是通常字节、字或双字。它们的功能如表 1-2 所示。

表 1-2 80386 指令系统新增位处理指令的功能

指令	功能
bts	将指定的位置 1
btr	将指定的位置 0
btc	将指定的位置反
bt	测试指定的位
bsf	从低到高扫描目标，如果全为 0 则 ZF 置 1，否则 ZF 置 0，并把为 1 的位的序号放入目的寄存器
bsr	从高到低扫描目标，如果全为 0 则 ZF 置 1，否则 ZF 置 0，并把为 1 的位的序号放入目的寄存器

13. 处理器控制与特权指令

处理器控制与特权指令包括 hlt、wait、esc、nop 和控制寄存器存取指令。控制寄存

器存取指令如下：

```
mov eax,cr0
mov cr0,eax
mov eax,dr0
mov eax,dr0
mov eax,tr6
mov tr6,eax
```

注意 crn 和 drn 等几个寄存器只能用 mov 指令来存取，其他如 push、pop、add 等指令都不能对它们执行存取操作。

14. 高级语言支持指令

80386 指令系统新增了 3 条高级语言支持指令：bound、enter、leave。

bound 用于检查数组是否越界，例如，下面的指令检查 esi 是否在 100 与 200 之间，如果不是，则产生异常 5：

```
mem   dd 100,200
......
bound esi,mem
```

enter 和 leave 指令用于子程序中产生和消去堆栈框架。关于堆栈框架，请参阅后面章节中的相关内容。

15. 系统设置和测试指令

系统设置和测试指令在 8086 中都没有，下面是对它们的描述：

```
clts    ；清除 cr0 中的 ts 位，这是特权指令，只能在 0 环执行
sgdt    ；保存 gdtr 的内容
sldt    ；保存 ldtr 的内容
sidt    ；保存 idtr 的内容
ltr     ；装入任务寄存器
str     ；保存任务寄存器
lar     ；把源选择子的访问权限装入目的寄存器
lsl     ；把源选择子的段界限装入目的寄存器
lgdt    ；装入 gdtr
lldt    ；装入 ldtr
lidt    ；装入 idtr
verr    ；检查选择子是否可读
verw    ；检查选择子是否可写
lmsw    ；装入 msw
smsw    ；保存 msw
arpl    ；调整 arpl
```

这些指令用作保护模式的支持。

第二章 保护模式程序设计

本章主要介绍保护模式下的汇编语言程序设计，包括保护模式的基本理论，如何进入保护模式，如何进行任务转换，分页内存管理及虚拟 86 方式等。

2.1 保护模式概述

从 80286 开始，X86 系列的 CPU 提供了保护模式。所谓保护模式，是指 CPU 提供各种机制对内存进行保护。为照顾专门为 8086 编写的程序的兼容性，又提供了实模式，它与以前的 8086 完全兼容，当计算机启动时即进入这种方式。要进入保护模式，必须先经过软件的切换。在后面将会详细讲述切换入保护模式的步骤。在保护模式下，CPU 的寻址方式与实模式不同。在实模式下，CPU 的寻址方式是段基址+段偏移。例如，指令序列：

```
mov     bx,B800H
mov     ds,bx
mov     bx,1000H
mov     ax,[bx]
```

则最后 ax 中放的是地址 B9000 中的内容。计算方式如下。

```
    B8000
+    1000
-----------
    B9000
```

但在保护方式下情形不同，因为这时段寄存器 CS、DS、ES、FS、GS、SS 的意义已经不同，它里面放的不再是段基地址，而是指向一个表中的某项的指针。这个表可以是全局描述符表（GDT）或局部描述符表（LDT）。另外还有一个“中断描述符表”（IDT），这里先不讲述。GDT 或 LDT 由寄存器 gdtr 或 ldtr 指出，它们已在前面讲过，这里不再重述。GDT 或 LDT 中的内容可以是描述符或门。

如图 2-1 所示，GDTR 指明 GDT 在物理内存中的位置及其界限长度。

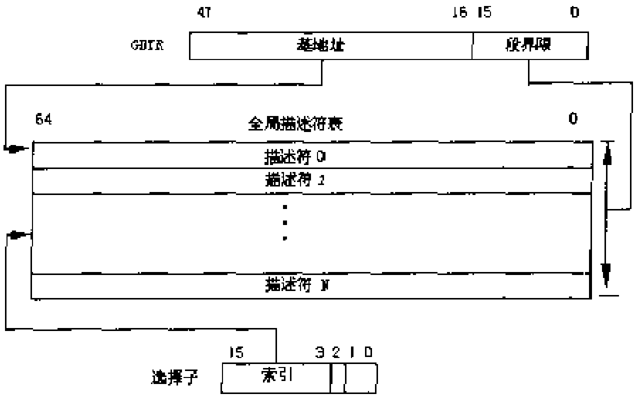


图 2-1

如果是 LDT，则还要转一个弯，因为 LDTR 是 16 位的，它也指向 GDT 中的一个描述符，然后再由这个描述符指出 LDT 的位置和界限长度。

“选择子”是 CS 或 DS、ES 等，它的第 0、1 位的值是特权级，标志着当前代码的特权或数据所属的特权级。

它的值可以是 0、1、2、3，数值越大，特权越低，反之越高。

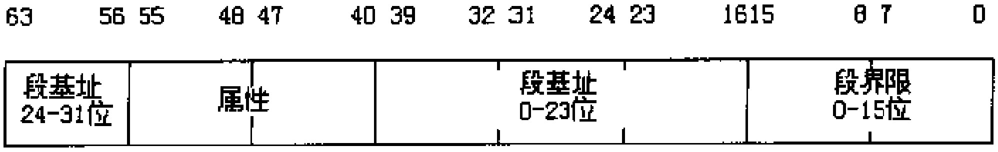
CS 寄存器中的低 2 位称为“当前特权级”。

当前特权级低的代码不能修改特权级高的数据，要修改当前特权级，必须经过特别的方法，不能只简单地修改 CS 中的值来更改。

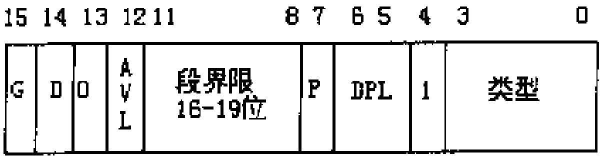
“选择子”第 2 位是选择 GDT 或 LDT 的标志，为 0 时表示 GDT，为 1 时是 LDT。第 3~15 位是描述符在 GDT 或 LDT 中的索引，即指向第几个描述符。

描述符分几种：普通段描述符、系统段描述符和门描述符。这里先介绍普通段描述符。

如图 2-2 所示，普通段描述符由属性字节、段基址、段界限组成。可以看到这几个部分都被很不自然地分成了几份，这是因为要兼容 80286 的程序。386 的段描述符是由 286 的段描述符扩展而成的。



a）普通段描述符



b）普通段描述符的属性字节

图 2-2

普通段描述符的属性字节的含义如下：

1）G 位，表示段界限是以字节（为 0 时）为单位还是以页（为 1 时）为单位。

2）D 位有多种意义：

① 当是代码段时，D 为 1 表示默认使用 32 地址及指令，为 0 时使用 16 位地址及指令；

② 当是堆栈段时，表示是用 32 位堆栈寄存器 ESP 还是 16 位的 SP。

③ 当是向低扩展的数据段时，D 为 1 时，上限为 4G，否则是 64K。

3）AVL 是用户自定义位。

4）P 位，存在位，表示段存在，即有效。

5）DPL 是描述符特权级，用以控制不同特权级对段的访问。

6）TYPE 是段类型，如表 2-1 所示。表中有一些不很常用的类型，Windows 大都没

有用到。

表 2-1 段类型

类型	描述
0	只读
1	只读，已访问
2	可读可写
3	可读可写，已访问
4	只读，向低扩展
5	只读，向低扩展，已访问
6	可读可写，向低扩展
7	可读可写，向低扩展，已访问
8	只执行
9	只执行，已访问
A	可读可执行
B	可读可执行，已访问
C	只执行，一致代码段
D	只执行，一致代码段，已访问
E	可读可执行，一致代码段
F	可读可执行，一致代码段，已访问

2.2 进入保护模式

进入保护模式首先要做的是建立系统表格，至少必须建立全局描述符表 GDT。这是因为分段的信息放在 GDT 中，而系统寻址必须通过分段的方式。即使是使用局部描述符表 LDT，LDT 的地址和界限等信息也放在 GDT 中。

建立起 GDT 及其他一些系统表格（如果有必要的话）后，就可以用 lgdtr 等指令来把 GDT 的地址及界限装入 GDTR 中，然后把 CR0 的 PE（保护模式功能）位置 1。这时需要用一个远程跳转指令 JMP XXXX: XXXX 来摆脱指令预取队列。这样，系统就进入保护模式，可以用保护模式下的段寻址方式进行寻址了。

下面的程序演示了进入保护模式的过程。

程序开始时建立 GDT 表。注意 GDT 表的第一项是不能使用的，即“选择子”的索引值为 0 且第 2 位也为 0 时，若用它来寻址则会引起异常。这种“选择子”称“空选择子”。

这个程序的可见效果是显示一串字符“Now Is In Protect Mode.”。但这不是通过调用 DOS 或 BIOS 中断来显示的。实际上，进入保护模式后，实模式下的中断就不能用了。本程序用直接写入显存地址的方法，把上面的字串写入地址 B8000H，因此，在 GDT 中建立了一个以 B8000H 为基地址的段。

程序要回到实模式跟进入保护模式的过程差不多，首先把 CR0 的 PE 位置 0，然后用一个长跳转脱指令预取队列。

程序的具体代码如下：

```
;************************
;文件：Enter.asm         *
;功能：演示进入保护模式  *
;************************

.386p
;----------------------
JUMP16 MACRO selector,offsetv
        DB 0EAH
        DW offsetv
        DW selector
        ENDM
;----------------------
Descriptor STRUC
    Limitl      dw 0
    basel       dw 0
    basem       db 0
    attrib      dw 0
    baseh       db 0
Descriptor ENDS

;================================
Data Segment use16
    gdt0 Descriptor <>          ;GDT 表开始

    DataSel = $-gdt0
    DataDes Descriptor <0ffffh,,,92H,>

    CodeSel = $-gdt0
    CodeDes Descriptor <0ffffh,,,98H,>

    VideoSel = $-gdt0
    VideoDes Descriptor <0ffffh,8000H.0BH,92H,>

    GdtLen   = $-gdt0

    GdtPtr    dw GdtLen
        dd 0
```

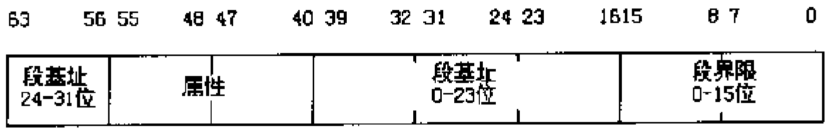
```
        Mess db 'Now Is In Protect Mode.',0
Data ends

;================================
Code Segment use16
        assume cs:Code,ds:Data
Start:
        xor eax,eax
        mov ax,Data
        mov ds,ax
        shl eax,4
        mov dword ptr [GdtPtr+2],eax
        mov DataDes.basel,ax            ;初始化数据段描述符
        shr eax,16
        mov DataDes.basem,al
        mov DataDes.baseh,ah
        xor eax,eax                     ;初始化代码段描述符
        mov ax,Code
        shl eax,4
        mov CodeDes.basel,ax
        shr eax,16
        mov CodeDes.basem,al
        mov CodeDes.baseh,ah
        lgdt qword ptr GdtPtr     ;Load GDT
        cli
        mov eax,cr0                     ;转到保护模式模式
        or eax,1
        mov cr0,eax
            JUMP16 CodeSEL,<offset Code16>
Code16:
        mov ax,DataSel
        mov ds,ax
        mov ax,VideoSel
        mov es,ax
        mov si,offset Mess
        mov di,80*10
        cld
        mov ah,1eh
```

```
Load:
      lodsb
      cmp al,0
      jz ReadyToReal
      stosw
      jmp Load
ReadyToReal:                    ;返回实模式
      mov eax,cr0
      and eax,0fffffffeh
      mov cr0,eax
      JUMP16 <seg Real>,<offset Real>
Real:
      sti
      mov ax,Data
      mov ds,ax
      mov es,ax
      mov ax,4c00h
      int 21h
Code ends
      end Start
```

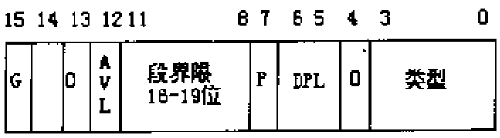
2.3 任务转换

Windows 是一个多任务系统。所谓多任务，是指系统在同一时间内运行多个任务。但我们知道，实际上 CPU 每一个确定的时刻只能执行一个指令流，要实现多任务，只能不断地在多个任务之间做切换。现在来看看如何进行任务切换。

要讲任务切换，就不得不先说说系统段描述符。系统段有两种：任务状态段 TSS 和局部描述符表 LDT。系统段描述符跟普通段描述符很相似，如图 2-3 所示。



a) 系统段描述符



b) 系统段描述符的属性字节

图 2-3

80X86 CPU 提供“任务状态段”来保存任务的状态。当任务切换时，用 call 指令指向一个任状态段，则 CPU 会保存当前的任务状态，如通用寄存器值、堆栈指针、标志寄存器的值等到当前的任务状态段中，并取出目标任务状态段的状态装入 CPU，执行新的任务状态段的任务，这样就完成了一次任务转换。

一般来说，分时操作系统会在定时器中断时判断是否要进行任务切换，操作系统查看当前进程或线程的时间片是否已用完，若是，则进行任务切换。

下面的程序演示用任务门进行任务切换的方法。

```
;**************************
;文件：Ttt.asm              *
;功能：任务转换实例         *
;**************************
.386p
JUMP16 MACRO selector,offsetv
         DB 0EAH
         DW offsetv
         DW selector
       ENDM
;----------------------
CALL16 MACRO selector ,offsetv
         DB 09AH
         DW offsetv
         DW selector
       ENDM
;----------------------
Descriptor STRUC
    limitl        dw 0
    basel         dw 0
    basem         db 0
    attributes dw 0
    baseh         db 0
Descriptor ENDS

;==============================
Data Segment use16
    gdt0 Descriptor <>
    DataSel = $-gdt0
    DataDes Descriptor <0ffffh,,,92H,>
    CodeSel = $-gdt0
```

```
CodeDes Descriptor <0ffffh,,,98H,>

Tss1Sel = $-gdt0                      ;Tss1 描述符
Tss1Des Descriptor <Tss1Len-1,,,89H,>
Tss2Sel = $-gdt0                      ;Tss1 描述符
Tss2Des Descriptor <Tss2Len-1,,,89H,>
Stack1Sel = $-gdt0
Stack1Des Descriptor <0,,,97H,>
Stack2Sel = $-gdt0+3
Stack2Des Descriptor <0,,,0F7H,>
Tss2CodeSel = $-gdt0+3
Tss2CodeDes Descriptor <0ffffh,,,0F8H,>
GdtLen   = $-gdt0
GdtPtr   dw GdtLen-1
         dd 0
;++++++++++++++++
Tss1 label byte
    dd 0
    dd 0        ;Stack Pointer for Ring 0
    dw 0,0
    dd 0        ;Stack Pointer for Ring 1
    dw 0,0
    dd 0        ;Stack Pointer for Ring 2
    dw 0,0
    dd 0              ;cr3
    dd 0              ;eip
    dd 0              ;eflags
    dd ?              ;eax
    dd ?              ;ecx
    dd ?              ;edx
    dd ?              ;ebx
    dd ?              ;esp
    dd ?              ;ebp
    dd ?              ;esi
    dd ?              ;edi
    dw 0,0            ;es
    dw 0,0            ;cs
    dw 0,0            ;ss
```

```
        dw 0,0                  ;ds
        dw 0,0                  ;fs
        dw 0,0                  ;gs
        dw 0,0                  ;ldt
        dw 0
        dw $+2
        db 0ffh
Tss1Len = $-Tss1
Tss2 label byte
        dd 0                    ;Link
        dd 0                    ;Stack Pointer for Ring 0
        dw 0,0
        dd 0                    ;Stack Pointer for Ring 1
        dw 0,0
        dd 0                    ;Stack Pointer for Ring 2
        dw 0,0
        dd 0                    ;cr3
        dw offset Tss2Begin,0                   ;eip
        dd 0                    ;eflags
        dd ?                    ;eax
        dd ?                    ;ecx
        dd ?                    ;edx
        dd ?                    ;ebx
        dd 1024                 ;esp
        dd ?                    ;ebp
        dd ?                    ;esi
        dd ?                    ;edi
        dw 0,0                  ;es
        dw Tss2CodeSel,0                ;cs
        dw Stack2Sel,0                  ;ss
        dw 0,0                      ;ds
        dw 0,0                      ;fs
        dw 0,0                      ;gs
        dw 0,0                      ;ldt
        dw 0
        dw $+2
        db 0ffh
Tss2Len = $-Tss2
```

```
;++++++++++++++++++++++++++++++
      OldSPSS dw 0,0
Data ends
;==============================
Stack1 Segment use16
      db 1024 dup (0)
Stack1 Ends
Stack2 Segment use16
      db 1024 dup (0)
Stack2 Ends
;==============================
Tss2Code Segment use16
      assume cs:Tss2Code
Tss2Begin:
      iretd
Tss2Code Ends
;==============================
Code Segment use16
      assume cs:Code,ds:Data
Start:
      mov ax,Data
      mov ds,ax
      xor eax,eax
      mov ax,Data
      shl eax,4
      mov dword ptr [GdtPtr+2].eax          ; GDT 地址
      xor eax,eax                           ;初始化数据段描述符
      mov ax,Code
      shl eax,4
      mov CodeDes.basel,ax
      shr eax,16
      mov CodeDes.basem,al
      mov CodeDes.baseh,ah
      mov ax,Data                           ;Tss1
      movzx eax,ax
      shl eax,4
      add eax,offset Tss1
      mov Tss1Des.basel,ax
```

```
shr eax,16
mov Tss1Des.basem,al
mov Tss1Des.baseh,ah
mov ax,Data                          ;Tss2
movzx eax,ax
shl eax,4
add eax,offset Tss2
mov Tss2Des.basel,ax
shr eax,16
mov Tss2Des.basem,al
mov Tss2Des.baseh,ah
mov ax,Stack1
movzx eax,ax                         ;Stack
shl eax,4
mov Stack1Des.basel,ax
shr eax,16
mov Stack1Des.basem,al
mov Stack1Des.baseh,ah
mov ax,Stack2
movzx eax,ax
shl eax,4
mov Stack2Des.basel,ax
shr eax,16
mov Stack2Des.basem,al
mov Stack2Des.baseh,ah
mov ax,Tss2Code                      ;Tss2Code
movzx eax,ax
shl eax,4
mov Tss2CodeDes.basel,ax
shr eax,16
mov Tss2CodeDes.basem,al
mov Tss2CodeDes.baseh,ah
lgdt qword ptr GdtPtr     ;Load GDT
cli
mov OldSPSS,sp
mov [OldSPSS+2],ss
mov eax,cr0                     ;Convert Into Protect Mode
or   eax,1
```

```
        mov cr0,eax
        JUMP16 CodeSEL,<offset Protect>
Protect:
        mov ax,DataSel
        mov ds,ax
        mov es,ax
        mov fs,ax
        mov gs,ax
        mov ax,Stack1Sel
        mov ss,ax
        mov sp,1024
        mov ax,Tss1Sel
        ltr ax
        CALL16 Tss2Sel,0                    ;Task Convert
        mov eax,cr0
        and eax,0fffffffeH
        mov cr0,eax
        JUMP16 Code,<offset Real>
Real:
        mov ax,data
        mov ds,ax
        lss sp,dword ptr OldSPSS
        sti
        mov ax,4C00H
        int 21H
Code ends
        end Start
```

2.4 分页内存管理

2.4.1 分页的功能

所谓分页，即是把内存分成固定大小的块（称为页），当程序要访问一个地址时，CPU根据系统设置的分页表格，把它映射到相应的物理块及其相应的偏移地址中。

例如，程序要访问地址 123456878H：

mov eax,[12345678]

假设这时 DS 指向的基地址为 0，则当没有启动分页机制时，这条指令的结果是 eax 中放的就是内存物理地址“12345678H”中的内容。但是，如果系统启动了分页，这时，

地址"12345678H"就会被CPU映射到别的地方去，而不一定是物理地址"12345678H"了。

这样做有什么实际意义呢？一个很容易想到的作用就是任务隔离。像Windows这样的多任务系统，为了要使各进程之间不互相干扰，就可以为不同的进程生成不同的逻辑地址到物理地址的映射表格（下面称"页表"）中。这样，即使一个错误的或恶意的进程使用了错误的指针，它也不可能访问到别的进程的地址空间，这就实现了任务的隔离。而这种隔离在如同DOS这种操作系统中是不可能实现的，一个恶意的进程可以破坏别的进程甚至整个操作系统。

另外，它还可以实施一些更灵活的访问控制。例如，页表的每一项都有一些位控制这些页分别对用户和系统是否可读，是否可写。这样，就使得用户进程可以执行系统提供的子程序，又不用担心用户进程改写它。

另外还有一个很常用的作用是方便实现虚拟内存。如果读者熟悉Windows编程，大概也知道Windows中每个进程有4GB的内存空间。但是，实际上机器中一般并没有这么多内存，那么操作系统又是怎样实现的呢？这是因为页表中每一项都有一个位指明这一个物理页是否真的在内存中，如果不在内存中，则对它访问时CPU就会产生异常。这时，就可以在异常处理程序中把页从硬盘上调入内存中，再执行相应的指令。

80386 CPU为分页功能增加了2个寄存器：CR2和CR3。CR3称为页目录表寄存器，CR2称为页异常线性地址寄存器。

为节省内存，80X86采用两级页表结构，如图2-4所示，CR3指向第一级页表，第一级页表的每一项指向第二级页表，第二级页表的每一项指向一个物理页。注意，若某一项的"存在"为0，则其指向的页表或物理页不存在。

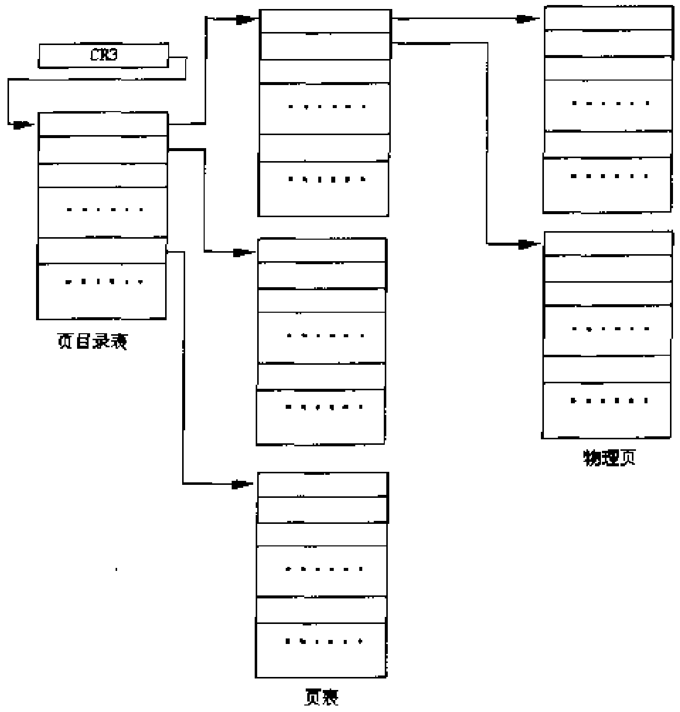


图2-4

页表目录或页表的项如图 2-5 所示。

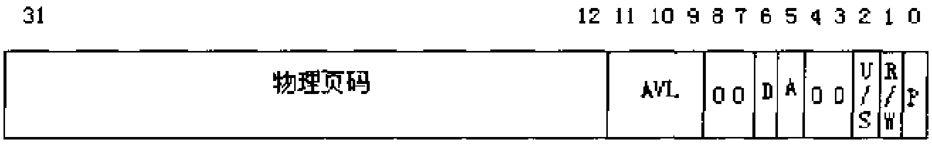


图 2-5

各位含义如下：

1）第 0 位是存在位，表示对应的物理页是否存在，P 为 1 表示存在。

2）第 1、2 位即 U/S 和 R/W 位。U/S 表示是用户（User）页还是系统（System）页；R/W 表示读（Read）和写（Write）权限。这两个位是表示这个页的存取权限。

3）第 3、4 位表示已访问过。

4）第 5、6 位为脏标志，用于标记页是否已经被写过。A 和 D 位都是为减少实现虚拟内存的换页操作而存在。

5）第 9、10、11 位（AVL）是软件自定义位，其意义由编程者定义。

6）12~31 位为物理页码，表示页表或物理页所在的物理页码。

图 2-4 中的页表或页目录表及物理页大小都是 1 页（4096 字节），每一项 4 字节，即共 1024 项。当一个逻辑地址要转换成物理地址时，其转换过程如图 2-6 所示。

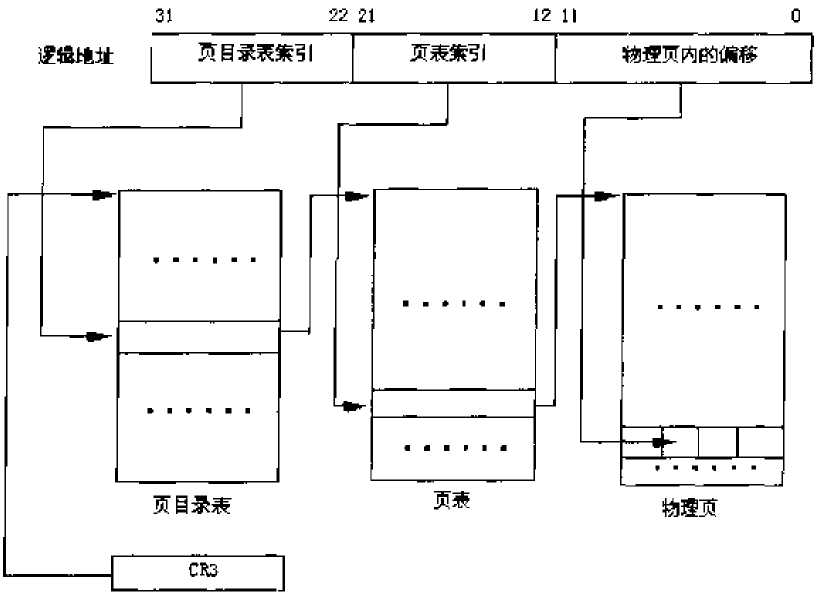


图 2-6

2.4.2　页保护

80386 分 4 个保护环，代表 4 个特权级。第 0、1、2 环称为系统特权级，第 3 环是用户特权级。

页表项和页目录表的第 1、2 位是页的访问权限位，它结合系统的特权级进行存取控

制。具体含义如下：

1）第 1 位是 R/W 位，当为 1 时，是可写可读页，当为 0 时，是只读页。80X86 系统的页保护中没有执行位，只要页是可读的，它就是可执行的。

2）第 2 位是 U/S 位，当为 1 时，表示是用户页，当为 0 时，是系统页。当某页是系统页时，不论 R/W 位是什么，用户特权级的代码都不能访问它；而当代码处于系统特权级时，则不论 R/W 位是什么，代码都有可以读写它。也就是说，R/W 位只对用户特权级的代码有效。

现在有一个问题是，页表项与页目录表项中都有 R/W 和 U/S 位，如果它们不相同，那怎么办呢？答案是做“与”运算，即页表项的这两个属性与页目录表的这两个属性做“AND”运算。换句话说，只有当页表项与页目录表项的两个属性都为 1 时，综合的属性才是 1。

当代码对页进行了越出它权限的操作，或它所访问的页根本不存在时，就会产生异常。页的异常代码是 0EH。这是一个故障类异常，进入异常时堆栈中的 CS：EIP 值指向产生异常的指令。

2.4.3 分页实例

下面是一个进入和脱离分页机制的例子。要使 CPU 分页机制生效，应将 CR0 的第 31 位置 1，并用一个跳转指令摆脱指令预取队列。脱离分页机制是将 CR0 第 31 位置 0，再执行一个跳转指令。

下面的程序分页使用后，向逻辑地址 1A0000H 写入一个字符串。但是，我们已把逻辑地址 1A0000 映射到物理地址 B8000H，这是显存的地址。因此，当写入时，会在屏幕上显示字符串“Now,Page Is OK!”。

```
;************************
;文件：Page.asm          *
;功能：演示分页内存管理  *
;************************

.386p
;----------------------
JUMP32 MACRO selector ,offsetv
        DB 0EAH
        DW offsetv
        DW 0
        DW selector
        ENDM
;----------------------
JUMP16 MACRO selector,offsetv
        DB 0EAH
```

```
        DW offsetv
        DW selector
        ENDM
;--------------------
Descriptor STRUC
    limitl      dw 0
    basel       dw 0
    basem       db 0
    attributes  dw 0
    baseh       db 0
Descriptor ENDS
;==============================
Data Segment use16

    gdt0 Descriptor <>
    DataSel = $-gdt0
    DataDes Descriptor <0ffffh,,,92H,>

    CodeSel = $-gdt0
    CodeDes Descriptor <0ffffh,,,98H,>

    ProtSel = $-gdt0
    ProtDes Descriptor <0ffffh,,,4098H,>

    PageSel = $-gdt0
    PageDes   Descriptor <0ffffh,0000,20H,92h,>

    VideoSel = $-gdt0
    VideoDes Descriptor <0ffffh,0000H,1AH,92H,>

    GdtLen   = $-gdt0

    GdtPtr   dw GdtLen
             dd 0

    PageOK db 'Now,Page Is OK!',0
Data ends
;==============================
```

```
Protect32 segment use32
      assume cs:Protect32
Begin:
      mov ax,PageSel
      mov es,ax

            cld                     ;初始化页表
            xor edi,edi
            mov eax,201000H
Again:
            or eax,1
            stosd
            add eax,1000H
            cmp edi,1000H
            jnz Again

            xor eax.eax
            mov edi,1000H
Again1:
            and eax,0FFFFF000H
            or eax,1
            stosd
            add eax,1000H
            cmp eax,200001H
            jnz Again1

            mov ecx,4
            mov eax,0b8000h
            or eax,1
            mov edi,1A0000H
            shr edi,22
            shl edi,2
            mov edi,es:[edi]
            and edi,0FFFFEH
            mov ebx,1A0000H
            shl ebx,10
            shr ebx,22
            shl ebx,2
```

```
        add edi,ebx
Again2: stosd
        add eax,1000H
        loop Again2

        mov eax,200000H
        mov cr3,eax
        mov eax,cr0                    ;分页
        or eax,80000000H
        mov cr0,eax
        jmp ShowMessage
ShowMessage:
        mov ax,DataSel
        mov ds,ax
        mov ax,VideoSel
        mov es,ax
        mov esi,offset PageOK
        mov edi,80*4
        mov ah,1eh
Load1:  lodsb
        cmp al,0
        jz UnPage
        stosw
        jmp Load1

UnPage: mov eax,cr0
        and eax,7FFFFFFFH
        mov cr0,eax
        jmp OK
OK:
        JUMP32 CodeSel,<offset ReadyToReal>
protect32 ends
;============================
Code Segment use16
    assume cs:Code,ds:Data
Start:
    xor eax,eax
    mov ax,Data
```

```
    mov ds,ax
    shl eax,4
    mov dword ptr [GdtPtr+2],eax
    mov DataDes.basel,ax          ;初始化数据段描述符
    shr eax,16
    mov DataDes.basem,al
    mov DataDes.baseh,ah
    xor eax,eax                   ;初始化代码段描述符
    mov ax,Code
    shl eax,4
    mov CodeDes.basel,ax
    shr eax,16
    mov CodeDes.basem,al
    mov CodeDes.baseh,ah
    xor eax,eax                   ;初始化 32 位保护模式代码段描述符
    mov ax,Protect32
    shl eax,4
    mov ProtDes.basel,ax
    shr eax,16
    mov ProtDes.basem,al
    mov ProtDes.baseh,ah
    lgdt qword ptr GdtPtr    ;Load GDT
    cli
    call EA20
    mov eax,cr0                   ;转到保护模式
    or eax,1
    mov cr0,eax
    JUMP16 ProtSel,0
ReadyToReal:
    mov eax,cr0
    and eax,0fffffffeh
    mov cr0,eax
    JUMP16 <seg Real>,<offset Real>
Real:
    call DA20
    sti
    mov ax,Data
    mov ds,ax
```

```
        mov es,ax
        mov ax,4c00h
        int 21h
;-----------------------------
EA20 PROC
        push ax
        in al,92h
        or al,2
        out 92h,al
        pop ax
        ret
EA20 ENDP
DA20 PROC
        push ax
        in al,92h
        and al,0fdh
        out 92h,al
        pop ax
        ret
DA20 ENDP
;----------------------
Code ends
        end Start
```

2.5 虚拟 86 方式

在 Windows 中，当要打开一个 MS-DOS 方式时，Windows 会打开一个窗口，用户可以在里面运行大多数的 DOS 程序，在命令行中输入命令，就像在 DOS 下一样工作。这一切都依靠 80386 处理器的一个工作方式：虚拟 86 方式。

虚拟 86 方式也处在保护模式下，这种模式的设计主要是为了兼顾软件的兼容性和保护模式的强大保护功能。处于虚拟 86 方式下的代码采用实方式下的寻址方式，即段寄存器乘 16 加偏移地址。

当标志寄存器 EFLAG 的 VM 位为 1 时，处理器即处于虚拟 86 模式。但不能简单地通过将 VM 位置 1 来切入 V86 方式。

进入 V86 方式的方法不止一途，可以通过任务切换，也可以通过用 iret 指令进入。虚拟 86 方式必须工作在特权级 3 上，而中断/异常处理程序是工作在 0 特权上的。因此，当在处理器工作在虚拟 86 方式又发生中断/异常时，处理器中断并离开虚拟 86 方式，进入中断/异常处理程序，这时，0 级堆栈中放有标志寄存器 EFLAGE 的值，如图 2-7 所示。

它的 VM 位是 1，当中断返回时会再载入 EFLAGE，再进入虚拟 86 方式。我们可以模拟一个中断的环境，之后执行 iret 指令，进入虚拟 86 方式。

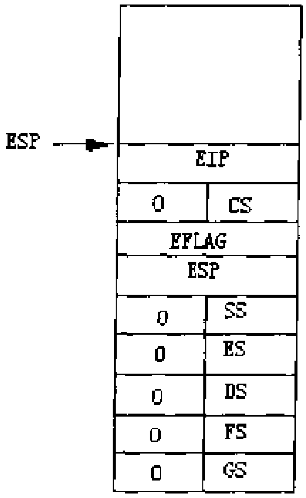


图 2-7

下面的例子是用任务转移的方式转入虚拟 86 方式，但其中的中断处理也说明了通过 iret 指令进入虚拟 86 方式的方法。

程序设置了二个任务状态段，其中一个 EFLAGE 中的 VM 位为 1，称为 TssV86。程序进入保护模式后，设置当前任务状态段，然后用一长跳转指令，它的目标是 TssV86，这就进入了 V86 模式，最后驻留程序退出。

这样，系统回到命令行方式，可以执行一般的 DOS 程序。但是，如果某些程序要用到一些特权指令或想再进入保护模式，则会引起异常，由于这个程序很简单，不能处理这么复杂的情形，因此这时可能会死机。

程序的中断处理方法很简单，就是把控制权返回给实模式下的中断处理过程。程序从内存地址 0 处的中断向量表取得相应的中断处理过程地址，并把返回地址设成相应的中断处理地址，并安排好供 DOS 下中断返回的堆栈，然后执行 iret 指令返回实模式下的中断处理过程。

```
;***********************************
;文件：V86.asm                     *
;用任务转移的方法进入 V86 方式     *
;***********************************
.386p
JUMP16 MACRG selector,offsetv
          DB 0EAH
          DW offsetv
          DW selector
          ENDM
;----------------------
```

```
Descriptor STRUC
        Limitl          dw 0
        basel           dw 0
        basem           db 0
        attrib          dw 0
        baseh           db 0
Descriptor ENDS
;---------------------
Gate STRUC
        Offsetl         dw 0
        Selector        dw 0
        Dcount          db 0
        Gtype           db 0
        Offseth         dw 0
Gate ENDS
;===============================
Data Segment use16
        gdt0 Descriptor <>
        DataSel = $-gdt0
        DataDes Descriptor <0ffffh,,,92H,>

        AllSel = $-gdt0
        AllDes Descriptor    <0ffffH,,,0F92H,>

        CodeSel = $-gdt0
        CodeDes Descriptor <0ffffh,,,98H,>

        TssV86Sel = $-gdt0                          ;TssV86 Descriptor
        TssV86Des Descriptor <TssV86Len-1,,,89H,>

        TssTSel = $-gdt0                            ;TSS Descriptor
        TssTDes Descriptor <TssTLen-1,,,89H,>
        GdtLen    = $-gdt0

        GdtPtr    dw GdtLen-1
                  dd 0
;++++++++++++++++++++++++++++++++++++++++++++++++
StackBuff db 1024 dup (0)
```

```
        StackTop label byte

        OldSPSS dw 0,0
;+++++++++++++++++++++++++++++++++++++++++++++++++
        TssTemp dd 0
                dd 0        ;Stack Pointer for Ring 0
                dw 0,0
                dd 0        ;Stack Pointer for Ring 1
                dw 0,0
                dd 0        ;Stack Pointer for Ring 2
                dw 0,0
                dd 0            ;cr3
                dd 0            ;eip
                dd 0            ;eflags
                dd ?            ;eax
                dd ?            ;ecx
                dd ?            ;edx
                dd ?            ;ebx
                dd ?            ;esp
                dd ?            ;ebp
                dd ?            ;esi
                dd ?            ;edi
                dw 0,0          ;es
                dw 0,0          ;cs
                dw 0,0          ;ss
                dw 0,0          ;ds
                dw 0,0          ;fs
                dw 0,0          ;gs
                dw 0,0          ;ldt
                dw 0
                dw 68H
                db 0ffh
        TssTLen = $-TssTemp

        TssV86      dd 0
                dd StackTop     ;Stack Pointer for Ring 0
                dw DataSel,0
                dd 0            ;Stack Pointer for Ring 1
```

```
                dw 0,0
                dd 0            ;Stack Pointer for Ring 2
                dw 0,0
                dd 0      ;cr3
                dw offset V86,0          ;eip
                dd 00023000H             ;eflags
                dd ?            ;eax
                dd ?            ;ecx
                dd ?            ;edx
                dd ?            ;ebx
                dd 0            ;esp
                dd ?            ;ebp
                dd ?            ;esi
                dd ?            ;edi
                dw Data,0       ;es
                dw Code,0       ;cs
                dw Data,0       ;ss
                dw Data,0       ;ds
                dw Data,0       ;fs
                dw Data,0       ;gs
                dw 0,0          ;ldt
                dw 0
                dw 68H
                db 65536/8 dup (0)
                db 0ffh
        TssV86Len = $-TssV86

;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    IdtPtr      dw IdtLen-1
                dd 0
    IdtTable label byte
        Off = offset IntProc
    REPT 256
        Gate<Off,CodeSel,0,0EEH,0>
        Off=Off+(All-IntProc)/256
    ENDM
    IdtLen = $-IdtTable
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
        DataLen = $-gdt0
Data ends

;================================
Code Segment use16
        assume cs:Code,ds:Data
IntProc:
        COUNT = 0
        REPT 256
            push word ptr COUNT
            push ax
            mov ax,offset All
            jmp ax
            COUNT = COUNT+1
        ENDM

All:
        pop ax
        push bp
        mov bp,sp
        add bp,4
        push eax
        push esi
        mov ax,AllSel
        mov ds,ax

        xor   eax,eax
        mov   esi,[bp+10h]
        mov   ax,[bp+0ch]
        shl   esi,4
        add   eax,esi    ;
        mov   esi,[bp+8]   ;flag
        mov   [eax-2],si
        mov   esi,[bp+4]   ;cs
        mov   [eax-4],si
        mov   esi,[bp+0]   ;ip
        mov   [eax-6],si
        xor   esi,esi
```

```
        mov    si,[bp-2]
        shl    esi,2
        lodsd
        mov    [bp+0],ax      ;ip
        shr    eax,10h
        mov    [bp+4],ax      ;cs
        mov    eax,[bp+0ch]
        sub    eax,6
        mov    [bp+0ch],eax ;sp

        mov eax,[bp+8]
        or eax,23000H
;       and eax,0fffffeffH
        and eax,0fffffCffH
        mov [bp+8],eax
        pop       esi
        pop       eax
        pop       bp
        add       sp,02h
        iretd

Start:
        mov ax,Data
        mov ds,ax
        xor eax,eax
        mov ax,Data
        shl eax,4
        mov dword ptr [GdtPtr+2],eax            ;GDT 地址
        mov ax,Data
        movzx eax,ax
        shl eax,4
        add eax,offset IdtTable
        mov dword ptr [IdtPtr+2],eax            ;IDT 地址
        mov DataDes.basel,ax                    ;初始化数据段描述符
        shr eax,16
        mov DataDes.basem,al
        mov DataDes.baseh,ah
        xor eax,eax                             ;初始化代码段描述符
```

```
        mov ax,Code
        shl eax,4
        mov CodeDes.basel,ax
        shr eax,16
        mov CodeDes.basem,al
        mov CodeDes.baseh,ah
        mov ax,Data                     ;TssV86
        movzx eax,ax
        shl eax,4
        add eax,offset TssV86
        mov TssV86Des.basel,ax
        shr eax,16
        mov TssV86Des.basem,al
        mov TssV86Des.baseh,ah
        mov ax,Data                     ;tssTemp
        movzx eax,ax
        shl eax,4
        add eax,offset TssTemp
        mov TssTDes.basel,ax
        shr eax,16
        mov TssTDes.basem,al
        mov TssTDes.baseh,ah

        lgdt qword ptr GdtPtr     ;Load GDT
        cli
        lidt qword ptr IdtPtr     ;Load IDT
        mov OldSPSS,sp
        mov [OldSPSS+2],ss
        mov eax,cr0               ;转到保护模式模式
        or eax,1
        mov cr0,eax
        JUMP16 CodeSEL,<offset Protect>

Protect:
        mov ax,TssTSel
        ltr ax
        clts
        JUMP16 TssV86Sel,0
```

```
V86:
        mov ax,data
        mov ds,ax
        lss sp,dword ptr OldSPSS
        sti
        MOV AX,3100H
        mov dx,offset start + DataLen
        shr dx,4
        add dx,12h
        INT 21H

Code ends
        end Start
```

第三章　用汇编语言编写 Win32 应用程序

本章介绍用汇编语言写 Win32 程序的方法、步骤以及所需要的基础知识。先讲述如何在 C/C++中插入汇编指令；接着讲述使用纯粹的汇编语言来写程序的方法，其中包括编写.EXE 可直接执行程序和动态链接库.DLL；最后介绍虚拟设备驱动程序（VxD）的编写。

3.1　在 C/C++中插入汇编指令

用汇编语言写 Windows 程序无非有两种形式：在高级语言中插入汇编指令和直接使用纯粹的汇编语言来写。前者要求宿主语言支持插入汇编指令；后者是一种比较极端的形式。为了充分表达汇编语言的特点，本书的大多数程序都采用后一种形式。

需要内嵌汇编指令的程序大多数都是一些速度要求较高或需要做底层操作的程序，因此一般都使用 C/C++语言来写。这是因为 C/C++语言本身也可以做一些底层操作并且编译后的程序速度会很快。

在 C/C++中插入汇编指令所需的关键字在不同的操作系统，不同的编译器中可能各不相同。Win32 平台上较流行的 C/C++编译器 Visual C++、Borland C++和 C++ Builder 支持用 asm、_asm 或__asm 这几个关键字中的 2、3 个来插入汇编指令。这几个关键字的作用是一样的。例如：

```
asm mov eax,0x12345678
_asm xor ebx,ebx
__asm mov esi,edi
```

也可写成：

```
_asm{
        mov eax,0x12345678
        xor ebx,ebx
        mov esi,edi
}
```

下面的程序显示一个 MessageBox 框，图 3-1 是程序的运行结果。

图 3-1

```
#include<windows.h>
int APIENTRY WinMain(HMODULE hMod,HMODULE prehMod,LPSTR lpCmd,int n)
{
```

```
    char *lpCaption ="标题";
    char *lpText    ="C/C++中插入汇编指令示例";
    __asm{
        push MB_OK
        push lpCaption
        push lpText
        push NULL
        call dword ptr MessageBoxA
    }

    return 0;
}
```

程序中，_asm 后面的大括号里的汇编指令相当于使用 C 语言的一条 API 调用：

MessageBoxA（NULL,lpText,lpCaption,MB_OK）;

四个 Push 指令把 MessageBoxA 所需的参数按从右到左的顺序压入栈中。MessageBoxA 在返回时自动调整堆栈，丢弃这些入栈的参数。

3.2 编写 Win32 汇编程序

3.2.1 Win32 汇编编程入门实例

现在我们来看一下单纯用汇编语言编写 Win32 程序的方法。下面仿照各种介绍编程语言的书籍，先写一个显示字符串“Hello World!”的程序作为入门。当然，Win32 的标准界面是图形界面，而不是字符界面，因此我们要把这个字符串放在一个 MessageBox 框里，如图 3-2 所示。

图 3-2

1. 源程序代码

源程序如下：

```
;**********************************
;Hello.asm                        *
;功能：显示一个 MessageBox 框       *
;**********************************
NULL    = 0                 ;常量定义
MB_OK = 0                   ;
```

```
.386p
.model flat                ;平坦内存模式
extrn MessageBoxA:proc     ;外部过程声明
extrn ExitProcess:proc     ;

.data
    Caption db 'Hello World!',0
    Text    db '欢迎来到 Win32 汇编的世界',0

.code
main:
    push MB_OK             ;MessageBoxA 的参数入栈
    push offset Caption    ;
    push offset Text       ;
    push NULL              ;
    call MessageBoxA
    push 0                 ;ExtiProcess 的参数入栈
    call ExitProcess

    end main
```

下面我们来逐行解释这个程序：

1）第一、二行是常量定义，将常量 MB_OK 和 NULL 都定义为 0。

2）第三行“.386p”是声明程序要用到 80386 保护模式的指令。当然，这个程序没用到保护模式下的专用指令，因此可以只写成.386（.386p 中的 p 是 protected mode——保护模式之意）。

3）第四行 model flat 是规定程序的内存模式。Win32 程序一般都使用平坦（flat）模式。

需要了解关于内存模式的更多内容，请看后面的章节。

4）第五、六行是说明函数 MessageBoxA 是一个外部函数，不在本文件中定义。

5）数据部分“.data”包含了两个字符串，字符串必须以 0 结尾。

6）代码部分“.code”所做的事是调用了两个 Win32 API：MessageBoxA 和 ExitProcess。

MessageBoxA 显示如图 3-2 所示的对话框。

ExitProcess 则退出进程。

调用之前先把 API 函数所需的参数压入栈中，它相当于用 C 语言写的调用：

MessageBoxA(NULL,lpText,lpCaption,MB_OK);

ExitProcess(0);

注意，参数以从右到左的顺序入栈。

2. 编译

虽然可以在命令行中用 Tasm32 和 Tlink32 命令来编译这个程序，但是，每次编译时都要输入长长的两串命令，非常麻烦。可以建立一个名为 MakeFile 文件，这样，只要在命令行提示符下打 Make 就可以了（当然，先要设置搜索路径到 Tasm 的 Bin 目录下）。

MainFile 文件的内容如下：

```
NAME = Hello
OBJS = $(NAME).obj
IMPORT=$(MAKEDIR)\..\lib\import32
$(NAME).EXE: $(OBJS) $(DEF)
   tlink32 /Tpe /aa /c    $(OBJS),$(NAME),, $(IMPORT),
.asm.obj:
tasm32 /ml $&.asm
```

文件中的 NAME、OBJS、IMPORT 是字符串常量，如 NAME 被赋值为"Hello"，后面凡遇到$(NAME)时被展开成 Hello。MAKEDIR 是 Make.exe 的所在目录。Import32.lib 是一个库文件，它是 Tasm 提供的，也可以用程序 ImpLib.exe 生成。

3.2.2　中级编程实例

程序 Hello.exe 是十分简单的，基本上没有什么用户界面。下面介绍一个相对而言稍微复杂的例子。这个程序显示一个对话框，如图 3-3 所示。对话框上有文字，有编辑控制，还有一个按钮。

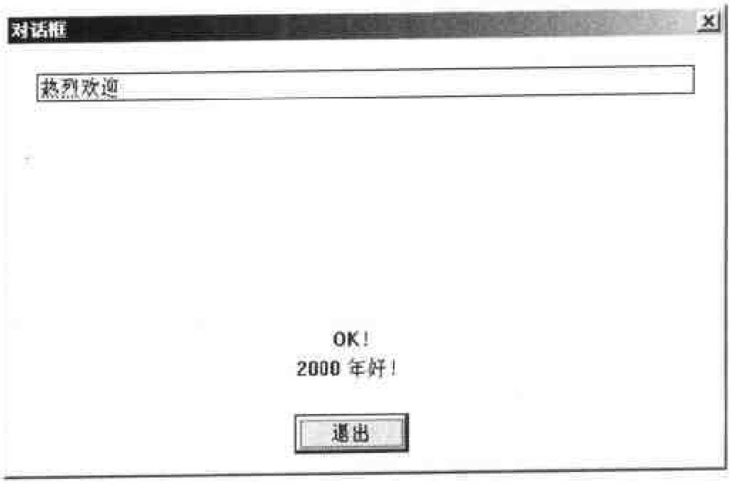


图 3-3

1. 源程序

源程序如下：

```
;*******************************************
;文件： ASM32.ASM                          *
;功能： 显示一个对话框                      *
;*******************************************
```

```
IDC_EDITTEXT   = 100              ;EDIT 控制的 ID
.386p
.model flat,stdCall               ;平坦内存模式，标准调用约定
include win32.inc                 ;包含有 Win32 部分常量定义的文件
extrn MessageBoxA:proc
extrn ExitProcess:proc
extrn GetModuleHandleA:proc
extrn DialogBoxParamA:proc
extrn EndDialog:Proc
extrn SetDlgItemTextA:proc

.data
    DialogName   db 'MyDialog',0
    CaptionFail  db '失败！',0
    TextFail  db 'API:DialogBoxParamA 调用失败！',0
    TEXT         db '热烈欢迎',0

.code
;主过程
Start:
    call GetModuleHandleA,0          ;取得模块句柄，返回结果在 eax 中
    ;调用 DialogBoxParamA 显示对话框
    call DialogBoxParamA,eax,offset DialogName,0,offset DialogProc,0
    cmp eax,0                        ;成功否？
    jnz  Exit                        ;成功则退出
    ;显示出错信息
    call MessageBoxA,0,offset TextFail,offset CaptionFail,MB_OK
Exit:
    call ExitProcess,0               ;退出进程
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;对话框消息处理过程
DialogProc proc uses ebx edi esi, hWnd:DWORD, wMsg:DWORD, wParam:DWORD,
lParam:DWORD
    cmp wMsg,WM_INITDIALOG
    jz WmInitDialog
    cmp wMsg,WM_CLOSE
    jz WmClose
    cmp wMsg,WM_COMMAND
    jz WmCommand
```

```
        jmp ExitProc
    WmInitDialog:                    ;对话框初始化处理
        call SetDlgItemTextA,hWnd,IDC_EDITTEXT,offset TEXT
        jmp ExitProc
    WmClose:                         ;用户按下了 Alt+F4 或单击右上角 X 按钮
        call EndDialog,hWnd,1        ;关闭对话框
        jmp ExitProc
    WmCommand:
        mov ebx,wParam
        cmp bx,IDOK                  ;是按下了“退出”键?
        jnz ExitProc
        call EndDialog,hWnd,1        ;关闭对话框
        jmp ExitProc
    ExitProc:                        ;退出消息处理
        xor eax,eax
        ret
    DialogProc endp

        end Start
```

2. 调用约定

.model 一行增加了一句 stdCall，表示“标准调用（Standard Call）”之意。这里涉及一个调用约定的问题。通常所用的调用约定是 C 调用约定和 Pascal 调用约定。

C 调用约定规定函数入口参数按从右到左的顺序入栈，并由调用者清理栈中参数。

Pascal 调用约定的入口参数则按从左到右的顺序入栈，并由被调用者清理栈中参数。

两者的函数返回值都是放在 eax（单精度）或 edx-eax（双精度）中。

例如下面的程序：

```
......
int result;
main()
{
......
    result=max(3,4);
.......
}
int max(int x,int y)
{
    return x>y?x:y;
}
```

- C 调用约定

若函数 max()是按 C 调用约定被编译的，则被译成下列汇编代码：

```
_main proc
      ......
      push 4
      push 3
      call _max
      add esp,8
      mov [result],eax
      ......
_main endp
_max proc
      push ebp
      mov ebp,esp
      mov eax,[ebp+12]
      mov ebx,[ebp+8]
      cmp eax,ebx
      jg   next
      mov eax,ebx
next:
      pop ebp
      ret
_max endp
```

- Pascal 调用约定

若函数 max()是按 Pascal 调用约定被编译的，则被译成下列汇编代码：

```
_main proc
      ......
      push 3
      push 4
      call _max
      mov [result],eax
      ......
_main endp
_max proc
      push ebp
      mov ebp,esp
      mov eax,[ebp+12]
      mov ebx,[ebp+8]
cmp eax,ebx
```

```
        jg   next
        mov eax,ebx
    next:
        pop ebp
        ret 8
    _max endp
```

- 标准调用约定

而 Win32 的标准调用约定则既不同于 C 调用约定也不同于 Pascal 调用约定，它采用 C 调用约定的入栈顺序和 Pascal 调用约定的调整栈指针方式，即函数入口参数按从右到左的顺序入栈，并由被调用者清理栈中参数。

上面例子按标准调用约定编译成汇编代码如下：

```
    _main proc
        ......
        push 4
        push 3
        call _max
        mov [result],eax
        ......
    _main endp
    _max proc
        push ebp
        mov ebp,esp
        mov eax,[ebp+12]
        mov ebx,[ebp+8]
        cmp eax,ebx
        jg   next
        mov eax,ebx
    next:
        pop ebp
        ret 8
    _max endp
```

3. 局部变量

另外，上面函数 max()的汇编代码都有一个 push ebp、mov ebp,esp 和 pop ebp 的框架（也可以改成指令 enter 和 leave）。或许会觉得这有些多余，因为指令 mov eax,[ebp+12] 可以写成 mov eax,[esp+12]。但是，其实这并非多余，因为：

1）函数可能要用到堆栈操作，中途会更改 esp（当然，这里没有）。

2）若 max()中有局部变量，则可在堆栈中通过下列指令序列创建和删除：

```
push ebp
mov ebp,esp
```

```
sub esp,n          ;创建局部变量 n 是局部变量所占字节数
......
add esp,n          ;删除局部变量
pop ebp
ret
```

例如：

```
int abc(int a)
{
    int i;
    i=a+1;
    return i;
}
```

编译成：

```
_abc proc
    push ebp
    mov ebp,esp
    sub esp,4                  ;创建局部变量 i，i 占 4 个字节
    mov ebx,[ebp+8]            ;取出参数 a
    inc ebx
    mov [ebp-4],ebx            ;传给 i
    mov eax,[ebp-4]            ;设置返回值
    add esp,4                  ;创建局部变量 i
    pop ebp
    ret 4
_abc endp
```

当然，这例子里的汇编代码可以再优化，这里仅仅是作为例子而已。

这样不但可以很快地创建局部变量（只需几条指令），而且这个函数是可重入的。事实上，若使用 Soft-Ice for Windows 或 Tr for Windows 之类的工具反汇编一下 Win32 的 API，会发现它们很多都是这样。

另外，还有一种“快速调用（fastcall）”是用寄存器来传递参数，因为与我们讨论的东西无关，这里不再细说。

4. 调用

程序中说明了.model stdcall，则可以简化一些子程序及对其调用的写法。首先是对调用的简化，例如，对 MessageBoxA 的调用的方法如下：

```
push MB_OK
push offset Caption
push offset Text
push NULL
call MessageBoxA
```

可以直接写成类似 C 语言的写法：

```
call MessageBoxA,NULL,offset Text,offset Caption,MB_OK
```

汇编程序会按 StdCall 的标准编译它。其次，StdCall 的声明可以简化子程序的写法。例如上面的汇编子程序_abc 可以写成：

```
_abc proc    a:DWORD
    LOCAL i              ;创建局部变量 i
    mov ebx,[a]          ;取出参数 a
    inc ebx
    mov [i],ebx          ;传给 i
    mov eax,[i]          ;设置返回值
    ret                  ;不需用 ret 4
_abc endp
```

5. 过程

主过程用 GetModuleHandleA 取得模块句柄（关于模块句柄，请看后面的章节）后，调用 DialogBoxParamA 来显示对话框。这个 API 要用到一个在资源文件 Asm32.rc 里定义的 Dialog 资源的名字或 ID（具体情况下面再讲述）。DialogBoxParamA 的另一个参数是这个对话框的消息处理函数，即是下面的 DialogProc，它的声明如下：

```
DialogProc proc uses ebx edi esi, hWnd:DWORD, wMsg:DWORD, wParam:DWORD, lParam:DWORD
```

其中 uses ebx edi esi 表示此过程最后要恢复 ebx、edi、esi 的内容，即编译程序会在过程的前后生成如下指令：

```
push ebx
push edi
push esi
.......
pop esi
pop edi
pop ebx
```

这是 Windows 的要求。

过程 DialogProc 开始时用一串 cmp wMsg,xxxx 判断响应的是什么消息，并跳转到相应的处理中，最后清除入栈的参数并退出。用户可以在这里加上自己要处理的消息如 WM_LBUTTONUP、WM_LBUTTONDBLCLK、WM_TIMER 等。

6. 资源文件

现在我们来看看资源文件 Asm32.rc，具体代码如下：

```
#define IDC_EDITTEXT   100                //EDIT 控制的 ID

MyIcon   ICON     Asm32.ico

MyDialog DIALOG   100,100,250,150
```

```
STYLE DS_MODALFRAME|WS_POPUP|WS_CAPTION|WS_SYSMENU
CAPTION "对话框"
{
        CTEXT "    OK!   ",-1,100,100,40,20
        CTEXT "2000 年好! ",-1,100,110,40,20
    PUSHBUTTON "退出",IDOK,100,130,40,14
    EDITTEXT          IDC_EDITTEXT,10,10,230,10,ES_MULTILINE|NOT
WS_TABSTOP
}
```

这个文件定义了两个资源：一个是图标资源 MyIcon，它对应于文件 Asm32.ico；另一个是对话框资源，它说明了对话框在屏幕中的位置，它的大小、风格、标题以及对话框里的窗口的位置、风格、大小、ID、名字等。

因为要加入资源文件，因此 MakeFile 文件必须改变，代码如下：

```
NAME = ASM32
OBJS = $(NAME).obj
RES  = $(NAME).res
IMPORT=$(MAKEDIR)\..\lib\import32
$(NAME).EXE: $(OBJS) $(DEF) $(RES)
  tlink32 /Tpe /aa /c $(OBJS),$(NAME),, $(IMPORT),,$(RES)
$(NAME).res: $(NAME).rc
   brc32 -r -fo$(RES) $(NAME).rc
.asm.obj:
   tasm32 /ml $&.asm
```

其具体参数含义不是这里的讲述重点，若读者有兴趣知道，请看 Tasm 所带的 help。一般的程序大多数都可按这个模式编译，以后若没有必要，我们将不再重复。

7. 窗口类和消息循环

上面的例子使用了对话框，使用对话框的好处是很容易编写，不需要自己建立窗口类和声明消息循环，但若要透彻了解 Windows 的工作过程，那么对窗口类和消息循环的理解是必不可少的。

- 注册窗口类

注册窗口类中存放着其实例窗口建立时的信息，如风格、窗口处理函数地址等。注册窗口类的 API 有 2 个：RegisterClass 和 RegisterClassEx。对 RegisterClass 的调用如下：

```
WNDCLASS struc
    style          dd ?        ；类风格
    lpfnWndProc    dd ?        ；消息处理函数指针
    cbClsExtra     dd ?        ；窗口类扩展字节数
    cbWndExtra     dd ?        ；窗口实例扩展字节数
    hInstance      dd ?        ；模块实例
```

```
        hIcon           dd ?            ; 图标句柄
        hCursor         dd ?            ; 指针句柄
        hbrBackground   dd ?            ; 背景画刷句柄
        lpszMenuName    dd ?            ; 菜单名指针
        lpszClassName   dd ?            ; 类名指针
          ends
        wc  WNDCLASS<>
    ……                                  ; 初始化 wc
        call  RegisterClassA,offset wc
```

对 RegisterClassExA 的调用完全相似，但它的参数不是 WNDCLASS 结构的指针，而是 WNDCLASSEX 结构的指针：

```
    WNDCLASSEX struc
        cbSize          dd ?            ; 结构大小
        style           dd ?
        lpfnWndProc     dd ?
        cbClsExtra      dd ?
        cbWndExtra      dd ?
        hInstance       dd ?
        hIcon           dd ?
        hCursor         dd ?
        hbrBackground   dd ?
        lpszMenuName    dd ?
        lpszClassName   dd ?
        hIconSm         dd ?            ; 小图标句柄
    ends
```

- 建立 Window

一般用高级语言建立窗口时有两个 API 可用：CreateWindowEx 和 CreateWindow，但是，实际上 CreateWindow 并不存在，Microsoft 在头文件中用一个宏使得你用 C/C++编程时，对 CreateWindow 的调用转向对 CreateWindowEx 的调用。CreateWindowEx 的调用如下：

call CreateWindowExA, extStyle, offset classname, offset winName ,style , x, y, width, height ,hWndParent,hMenu,hInst,lpData

其中：

extStyle 是扩展风格，有很多值可以设置。

classname 是注册的窗口类或系统提供的窗口类。

winName 是窗口的名字，它又会被设成窗口的标题。

style 是窗口风格。

x,y 窗口左上角的位置。

width，height 是窗口的宽和高。

hWndParent，父窗口的句柄。

hMenu，菜单句柄。

hInst 是作为模块实例句柄。

lpData，指向一个 CREATESTRUCT 结构，当 WM_CREATE 发出时，把它放在 lparam 参数中。

- 消息循环和窗口处理过程

消息循环是不停接收消息和发送消息的处理函数过程。每一个建立了窗口的线程都应该建立一个消息循环。消息循环一般用到以下几个 API：

1）用 GetMessageA 取得消息。

```
POINT struc
    x     dd ?
    y     dd ?
    ends
MSG struc
    hwnd     dd ?          ；窗口句柄
    message  dd ?          ；消息
    wParam   dd ?          ；消息参数 1
    lParam   dd ?          ；消息参数 2
    time     dd ?          ；消息发送的时间
    pt       POINT<>       ；消息发送时指针的位置
ends
msg      MSG<>
……
call GetMessageA,offset msg,hwnd,lowMsg,hiMsg
```

其中：

msg 是一个结构指针，存放消息和附加信息。

hwnd 是指明要接收的消息的窗口句柄，若为 NULL 则接收所有属于本线程的窗口的消息和用 PostThreadMessage 发送的消息。

lowMsg 和 hiMsg 是要接收消息的最小值和最大值。

2）用 TranslateMessage 把 GetMessageA 得到的虚拟键的消息转换为字符消息。

```
call TranslateMessage,offset msg
```

3）用 DispatchMessageA 分发消息。这个 API 的作用是查找 MSG 结构中的 hwnd 成员标志的窗口，并调用它的窗口处理函数：

```
call DispatchMessageA,offset msg
```

消息处理函数处理发给窗口的消息，它是一个标准调用约定的过程。它根据得到的消息进行相应的处理，但是，你可能并不想处理所有的消息，这时可以调用默认的消息处理函数 DefWindowProcA，并把消息处理函数得到的参数传给它：

```
call DefWindowProcA,handle,wMsg,wParam,lParam
```

● 实例

下面是一个注册窗口类并建立窗口的消息循环的例子，程序注册了一个叫“MyClass”的窗口类，并建立窗口。

窗口处理函数只处理了三个消息，分别是：

WM_CREATE。

WM_DESTROY。

WM_LBUTTONDOWN。

WM_CREATE 只是返回 0；WM_DESTROY 则用 PostQuitMessage 告诉系统线程要结束了；WM_LBUTTONDOWN 则弹出一个 MessageBox 框，如图 3-4 所示。

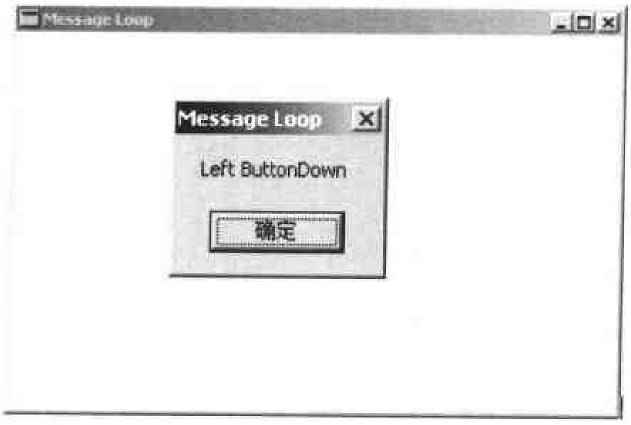


图 3-4

```
;*********************************
;MsgLoop.asm                     *
;功能：演示注册 window 类及      *
;      建立 Window 和消息循环    *
;*********************************
.386p
.model flat,stdcall
include win32.inc

extrn GetMessageA:proc
extrn TranslateMessage:proc
extrn DispatchMessageA:proc
extrn ShowWindow:proc
extrn UpdateWindow:proc
extrn CreateWindowExA:proc
extrn RegisterClassA:proc
extrn DefWindowProcA:proc
extrn MessageBoxA:proc
extrn ExitProcess:proc
```

```
extrn LoadIconA:proc
extrn LoadCursorA:proc
extrn GetStockObject:proc
extrn GetModuleHandleA:proc
extrn PostQuitMessage:proc

;结构定义
POINT struc
    x dd ?
    y dd ?
    ends
MSG struc
    hwnd        dd ?
    message     dd ?
    wParam      dd ?
    lParam      dd ?
    time dd ?
    pt          POINT<>
    ends

WNDCLASS struc
    style           dd ?
    lpfnWndProc     dd ?
    cbClsExtra      dd ?
    cbWndExtra      dd ?
    hInstance dd ?
    hIcon           dd ?
    hCursor         dd ?
    hbrBackground   dd ?
    lpszMenuName    dd ?
    lpszClassName   dd ?
    ends
.data ;数据
    Text        db 'Left ButtonDown',0
    myclassname db 'MyClass',0          ;窗口类名
    caption     db 'Message Loop',0     ;窗口标题
    hInst       dd ?
    hWnd        dd ?
```

```
        align 4         ;四字节对齐
        msg MSG<>                               ;消息结构
        wc  WNDCLASS<>                          ;窗口类结构

    .code           ;代码
    main:
        call  GetModuleHandleA,0                ;取模块句柄
        mov [hInst],eax

        mov wc.lpszClassName,offset myclassname
        mov eax,[hInst]
        mov wc.hInstance,eax
        mov wc.lpfnWndProc,offset WinMsgProc
        call  LoadIconA,0,IDI_APPLICATION
        mov wc.hIcon,eax
        call  LoadCursorA,0,IDC_IBEAM
        mov wc.hCursor,eax
        mov wc.lpszMenuName,0
        call  GetStockOhject,WHITE_BRUSH
        mov wc.hbrBackground,eax
        mov wc.style,CS_HREDRAW or CS_VREDRAW or CS_GLOBALCLASS
        mov wc.cbClsExtra,0
        mov wc.cbWndExtra,0
    ;注册窗口类
        call  RegisterClassA,offset wc
    ;建立主窗口
        call        CreateWindowExA,0,offset myclassname,offset caption,\
                                WS_OVERLAPPEDWINDOW,    \
                                100,100,450,300,0,0,    \
                                [hInst],0
        mov         [hWnd],eax

        call        ShowWindow,[hWnd],1
        call        UpdateWindow,[hWnd]
    ;消息循环
    MsgLoop:
        call        GetMessageA,offset msg,0,0,0
        or          eax,eax
        jz          Exit
```

```
        call    TranslateMessage,offset msg
        call    DispatchMessageA,offset msg
        jmp     MsgLoop
    Exit:
        call ExitProcess,0        ;退出进程
;************************************************************************
    ;主窗口过程
    WinMsgProc      proc    uses    ebx    edi    esi,    handle:DWORD,    wMsg:DWORD,
wParam1:DWORD,    lParam1:DWORD
        cmp     wMsg,WM_CREATE
        jz      create
        cmp     wMsg,WM_DESTROY
        jz      destory
        cmp     wMsg,WM_LBUTTONDOWN
        jz      lbuttondown
        jmp     default
    create:             ;窗口建立
        xor     eax,eax
        ret
    destory:            ;窗口注销
        call    PostQuitMessage,0
        xor     eax,eax
        ret
    lbuttondown:        ;鼠标单击
        call    MessageBoxA,handle,offset Text,offset caption,MB_OK
        xor     eax,eax
        ret
    default:            ;默认窗口处理过程
        call    DefWindowProcA.handle,wMsg,wParam1,lParam1
        ret
    WinMsgProc  endp
        end main
```

8. 关于 UNICODE

在上面的例子中我们可以看到，在调用一些 Windows API，要在后面加一个字符“A”，例如，对 MessageBox 函数，我们使用的是 MessageBoxA。但在高级语言如 C/C++等编程时并没有这种需要，这是为什么呢？这是由于 Microsoft 分别为这些 API 写了 ANSI 版本和 UNICODE 版本。

UNICODE 是一种统一的双字节字符集，它每一个字节都是 16 位的，不管它是一个英文字母还是一个汉字，或是日文，或是其他的国家的文字。由于 16 位宽的字符可以有

65536 个不同的字符，而不是 8 位 ANSI 的 256 个，因此，它可以把多种国家的语言用同一种编码来表示。

这种统一的字符编码对程序的编写和用户都是很有好处的。例如，你要编写一个程序的多个语言版本，若是使用了 UNICODE，则只需将相应的字符串转换成对应的国家的文字即可，而不需转换工作的环境；对于用户也一样，当他要在同一个机器上查看不同语言的文本或运行使用不同语言的程序时，就不需要如 RichWin 或南极星之类的内码转换工具了。当然，这要求他查看的文本或所用的程序是使用 UNICODE 的才行。例如，在 Windows 2000 的实用程序 Notepad.exe 中，用"另存为"菜单选项，出现的文件对话框中你可以把文件保存为 UNICODE，如图 3-5 所示。

图 3-5

回到我们 API 名字的问题上，对于 MessageBox，其实有两种版本：MessageBoxA 和 MessageBoxW。字符"A"和"W"分别是 ANSI 和 WIDTH 的缩写，即 ANSI 字符集和宽字符集。

很多 API 都有相类似的两个版本，如 SetWindowTextA 和 SetWindowTextW，又如 GetClassNameA 和 GetClassNameW。这些 API 一般都是它们的参数或返回值中有跟 UNICODE 有关的东西，很多是由于参数中有一个或多个字符串，如 MessageBox。也有一些参数中虽然没有字符串，但也会与 UNICODE 有关，甚至只是某些情况下与 UNICODE 有关，例如 SetWindowLongA 和 SetWindowLongW 以及 SendMessageA 和 SendMessageW 等等。在高级语言中，我们一般不需要注意这些细节，以 C/C++这例，Microsoft 在它给出的头文件中用宏定义和条件编译语句把这些函数调用映射到默认的版本中去。

UNICODE 在 Windows NT 中获得支持。Windows 95/98 虽然也有 SetWindowLongW、SendMessageW 这些 API，但它们绝大多数都只是返回错误，这时用 GetLastError 函数会得到 ERROR_CALL_NOT_IMPLEMENTED 的返回值。

为了通用性方面考虑，也为了编写的方便，本书的程序全部采用 ANSI 版本的 API。

9. 关于 Windows 版本

这本书的程序中有不少是与平台相关的，有的程序不能在 Windows NT 下运行，有的又只能在 Windows NT 下运行，但为节省篇幅，减小程序的长度以利于读者读懂它们，我们并没有在程序中加上版本检查。在这里，我们给出检查 Windows 版本的例程。

检查 Windows 版本可以使用 GetVersion 或 GetVersionExA，其中 GetVersionExA 提供更强大的功能，调用方式如下：

```
OSVERSIONINFO struc
    dwOSVersionInfoSize   dd ?              ; 本结构的大小
    dwMajorVersion        dd ?              ; Windows 主版本号
    dwMinorVersion        dd ?              ; Windows 次版本号
    dwBuildNumber         dd ?              ; 内部版本号
    dwPlatformId          dd ?              ; 指明平台
    szCSDVersion          db 128 dup (?)    ; 附加信息
ends
    OSVI     OSVERSIONINFO<>

    mov   OSVI.dwOSVersionInfoSize,size OSVERSIONINFO
    call GetVersionExA,offset OSVI
    or eax,eax
jz Error
```

结构 OSVERSIONINFO 的其他成员都一目了然，只有其中 dwPlatformId 可以是以下几个常量之一：

VER_PLATFORM_WIN32s，被定义为 0，表示 Windows 3.1 上的 Win32 平台。

VER_PLATFORM_WIN32_WINDOWS，定义为 1，表示 Winodws 95 或 98。

VER_PLATFORM_WIN32_NT，定义为 2，表示 Windows NT。

这个 API 在 Windows NT 2000 或以后的版本中可以用一个扩展的结构做参数来代替 OSVERSIONINFO 结构，定义如下：

```
OSVERSIONINFOEXA struc
    dwOSVersionInfoSize   dd ?              ; 本结构的大小
    dwMajorVersion        dd ?              ; Windows 主版本号
    dwMinorVersion        dd ?              ; Windows 次版本号
    dwBuildNumber         dd ?              ; 内部版本号
    dwPlatformId          dd ?              ; 指明平台
    szCSDVersion          db 128 dup (?)    ; 附加信息
    wServicePackMajor     dw ?              ; ServicePack 主版本号
    wServicePackMinor     dw ?              ; ServicePack 次版本号
    wReserved             dw 2 dup (?)      ; 保留
ends
```

程序在 Windows98 和 Windows 2000 中的运行结果如图 3-6 所示。

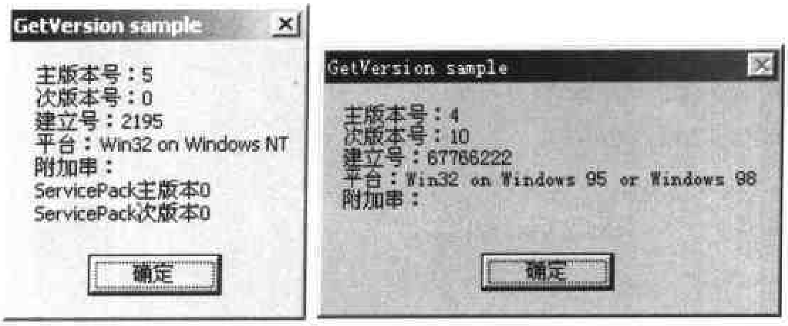


图 3-6

```
;*****************************
;文件：Version.asm          *
;作用：显示 Windows 版本信息  *
;*****************************
NULL   =0                    ;常量定义
MB_OK = 0
.386p
.model flat,stdcall
extrn GetVersion:proc
extrn GetVersionExA:proc
extrn MessageBoxA:proc
extrn ExitProcess:proc
extrn _wsprintfA:proc

VER_PLATFORM_WIN32s               =  0
VER_PLATFORM_WIN32_WINDOWS        =  1
VER_PLATFORM_WIN32_NT             =  2
OSVERSIONINFO struc
      dwOSVersionInfoSize    dd ?
      dwMajorVersion         dd ?
      dwMinorVersion         dd ?
      dwBuildNumber          dd ?
      dwPlatformId           dd ?
      szCSDVersion db 128 dup (?)
ends

OSVERSIONINFOEXA struc
      dwOSVersionInfoSize    dd ?
      dwMajorVersion         dd ?
```

```
        dwMinorVersion          dd ?
        dwBuildNumber           dd ?
        dwPlatformId            dd ?
        szCSDVersion db 128 dup (?)
        wServicePackMajordw ?
        wServicePackMinor       dw ?
        wReserved           dw 2 dup (?)
    ends
    .data
        OSVI    OSVERSIONINFO<>
        OSVIEX OSVERSIONINFOEXA<>
        Caption db 'GetVersion sample',0
        Text db 500 dup (?)
        Param    db '主版本号：%d',13,10
             db '次版本号：%d',13,10
             db '建立号：%d',13,10
             db '平台：%s',13,10
             db '附加串：%s'
        zero db 0,13,10
             db 'ServicePack 主版本%d',13,10
             db 'ServicePack 次版本%d',13,10
             db 0

        PF_31       db 'Win32s on Windows 3.1',0
        PF_9X       db 'Win32 on Windows 95 or Windows 98',0
        PF_NT       db 'Win32 on Windows NT',0
        UNKNOW      dd '未知',0
        ErrorText db '发生错误',0
    .code
    main:
        mov    OSVI.dwOSVersionInfoSize,size OSVERSIONINFO
        call GetVersionExA,offset OSVI
        or eax,eax
        jz Error

        mov ebx,OSVI.dwPlatformId
        cmp ebx,VER_PLATFORM_WIN32s
        jnz @1
        lea ebx,PF_31
```

```
        jmp MakeText
    @1:
        cmp ebx,VER_PLATFORM_WIN32_WINDOWS
        jnz @2
        lea ebx,PF_9X
        jmp MakeText
    @2:
        cmp ebx,VER_PLATFORM_WIN32_NT
        jnz @3
        lea ebx,PF_NT
        jmp NT5GetExt
    @3:
        lea ebx,UNKNOW
    MakeText:
        call _wsprintfA, offset Text, offset Param, OSVI.dwMajorVersion,
OSVI.dwMinorVersion, OSVI.dwBuildNumber, ebx, offset OSVI.szCSDVersion
        add esp,7*4
        jmp Show

    NT5GetExt:
        cmp OSVI.dwMajorVersion,5
        jb  MakeText
        mov  OSVIEX.dwOSVersionInfoSize,size OSVERSIONINFOEXA
        call GetVersionExA,offset OSVIEX
        or eax,eax
        jz Error
        lea ebx,PF_NT
        mov byte ptr [zero],' '
        mov cx,OSVIEX.wServicePackMajor
        mov dx,OSVIEX.wServicePackMinor
        movzx ecx,cx
        movzx edx,dx
        call _wsprintfA, offset Text, offset Param, OSVIEX.dwMajorVersion,
OSVIEX.dwMinorVersion, OSVIEX.dwBuildNumber, ebx, offset OSVIEX.szCSDVersion, ecx,
edx
            add esp,9*4
    Show:
        call MessageBoxA,NULL,offset Text,offset Caption,MB_OK
    Exit:
```

```
        call ExitProcess,0
    Error:
        call MessageBoxA,NULL,offset ErrorText,offset Caption,MB_OK
    jmp Exit
    end main
```

3.2.3 编写动态链接库

Win32 动态链接库 DLL（Dynamic Link Library）是应用程序的扩展，它由应用程序通过调用 Windows API LoadLibraryA 和 FreeLibrary 动态地载入和卸载。当一个进程启动时，在缺省的状态下，它会加载 Kernel32.dll、User32.dll 和 Gdi32.dll 等几个动态链接库，但是，有时候需要自己加载一些动态链接库（比如说是用户自己写的 DLL）。

动态链接库也分开数据（.data）和代码（.code）。要注意的是，当有多个进程把同一个 DLL 映射入自己的地址空间时，Windows 并不会复制多份的代码和数据，而只会通过 CPU 的页映射机制，把原有的代码和数据映射进自己的地址空间中。但是，当其中某一进程对一些数据作了更改时，Windows 就会复制一份对应页的数据给相应的进程，这样，就可以最大限度地节省内存，同时可以让各进程觉得自己有一份独立的拷贝。

下面是一个很简单的 DLL 的例子。它里面的函数 MsgBox 封装了 MessageBoxA 调用，使它只需要一个参数即可。

```
.386P
.Model Flat ,StdCall
include win32.inc
extrn MessageBoxA:PROC
public MsgBox
.Data
    Caption   db '动态链接库',0
.Code
DllMain:
    mov eax,1
    ret 0ch
MsgBox proc DWORD Text
    call MessageBoxA,0,Text,offset Caption,MB_OK
    ret
MsgBox Endp
    end DllMain
```

要编译这个文件，makefile 要作一些更改，具体如下：

```
NAME = DllDemo
OBJS = $(NAME).obj
DEF  = $(NAME).def
```

```
IMPORT=$(MAKEDIR)\..\lib\import32
$(NAME).EXE: $(OBJS) $(DEF)
  tlink32 /Tpd /aa /c $(OBJS),$(NAME),, $(IMPORT), $(DEF)
.asm.obj:
   tasm32  /ml $&.asm
```

以上的修改主要是增多了一个 DEF 文件的定义，并在 tlink32 中加上$(DEF)；同时将链接选项/Tpe 改成/Tpd，表示生成动态链接库。DllDemo.def 文件的代码如下：

```
NAME DllDemo
DESCRIPTION  'Win32 Dll Program'

CODE  PRELOAD MOVEABLE DISCARDABLE
DATA  PRELOAD MOVEABLE MULTIPLE

EXETYPE  WINDOWS
EXPORTS    MsgBox  @1
```

这个文件中，其他都不是很重要，最主要的是最后一句：EXPORTS MsgBox @1，用于声明这个 DLL 有一个引出函数可供外部调用，名叫 MsgBox，引出顺序号是 1。若有多个引出函数，则是@2、@3 等等。

现在，我们来测试这个 DLL。可以写一个程序来测试它：

```
;*************************
;文件：Test.asm          *
;功能：测试动态链接库    *
;*************************
.386P
Locals
.Model Flat ,StdCall
include win32.inc
extrn    ExitProcess:PROC
extrn    MessageBoxA:PROC
extrn    LoadLibraryA:PROC
extrn     FreeLibrary:PROC
extrn     GetProcAddress:Proc
.Data
    DllFileName  db '..\DllDemo\DllDemo.dll',0
    ProcName     db 'MsgBox',0
    TextSuc      db '动态链接库安装成功!',0
    TextErr      db '动态链接库链接失败或函数找不到',0
    TextExt      db '退出进程',0
    Caption      db 'Test',0
```

```
        TextDll     db '测试 DLL',0
        hDll        dd ?

    .Code
    Main:
        call LoadLibraryA,offset DllFileName        ;把 DLL 映射入进程空间
        or   eax,eax                                ;成功否?
        jz   Error
        mov hDll,eax
        call MessageBoxA,0, offset TextSuc,offset Caption,MB_OK
        call GetProcAddress,[hDll],offset ProcName      ;取 MsgBox 的地址
        or eax,eax
        jz Error
        call eax,offset TextDll                         ;调用 MsgBox
        call FreeLibrary,[hDll]                         ;释放 DLL
        jmp Exit
    Error:   call MessageBoxA,0, offset TextErr,offset Caption,MB_OK

    Exit:call MessageBoxA,0, offset TextExt,offset Caption,MB_OK
        ret 10H
    end Main
```

后面要释放 DLL 的原因是 Windows 对 DLL 的引用数有一个计数，只有当计数为 0 时，Windows 才从内存中释放 DLL。

程序运行结果如图 3-7 所示。

（a） （b） （c）

图 3-7

程序载入 DllDemo.dll 成功后，显示图 3-7（a）；然后取得 DllDemo.dll 中的引出函数 MsgBox，并调用它，传给 MsgBox 的是指向字符串“测试 DLL”的指针。调用 MsgBox 的结果是图 3-7（b）；最后释放 DLL，然后显示图 3-7（c）并退出进程。

3.2.4 编写控制台应用程序

所谓控制台应用程序，是指有 DOS 界面的 Windows 程序。它本身是 Windows 程序，不能在 DOS 下运行，但它具有 DOS 的字符界面。

下面是控制台程序的一个简单例子：

```
;**********************************
;Console.asm                     *
;功能：演示控制台应用程序的编写   *
;**********************************
.386p
.model flat,stdcall        ;平坦内存模式
NULL   = 0         ;常量定义
MB_OK = 0          ;
GENERIC_READ            equ  80000000H
GENERIC_WRITE           equ  40000000H
FILE_SHARE_READ         equ  00000001H
FILE_SHARE_WRITE        equ  00000002H
CONSOLE_TEXTMODE_BUFFER          equ  1
INVALID_HANDLE_VALUE             equ  -1

extrn WriteConsoleA:proc     ;外部过程声明
extrn SetConsoleTitleA:proc
extrn CreateConsoleScreenBuffer:proc
extrn CloseHandle:proc
extrn SetConsoleActiveScreenBuffer:proc
extrn MessageBoxA:proc
extrn ExitProcess:proc    ;

.data
     Caption db 'Hello World!',0
     Text     db '欢迎来到 Win32 汇编的世界',0
     ErrText   db 'Error',0
     Written dd ?
     hCS dd ?
.code
main:
     call SetConsoleTitleA ,offset Caption              ;改变窗口标题
;建立控制台的 screen buffer
     call CreateConsoleScreenBuffer, GENERIC_WRITE, FILE_SHARE_WRITE,NULL,
CONSOLE_TEXTMODE_BUFFER,NULL
     cmp   eax,INVALID_HANDLE_VALUE               ;建立失败？
     jz    Error
     mov   [hCS],eax
```

```
        call SetConsoleActiveScreenBuffer,eax          ;设置有效的制台 screen buffer
        ;写控制台
        call WriteConsoleA,[hCS],offset Text,offset ErrText - offset Text,offset Written,NULL
        call CloseHandle,[hCS]                         ;关闭句柄
        call ExitProcess,0
Error:
        call MessageBoxA,NULL,offset ErrText,offset Caption,MB_OK
        call ExitProcess,1
end main
```

这个程序的 makefile 也要作改变，只要把 Hello.asm 的 makefile 中如下语句：

```
$(NAME).EXE: $(OBJS) $(DEF)
  tlink32 /Tpe /aa /c    $(OBJS),$(NAME),, $(IMPORT),
```

改成：

```
$(NAME).EXE: $(OBJS) $(DEF)
  tlink32 /Tpe /ap /c    $(OBJS),$(NAME),, $(IMPORT),
```

即把参数/aa 改成/ap。

控制台应用程序和图形界面程序由 Windows 的两个不同的子系统支持（Windows NT 还支持 POSIX 子系统）。

如果用户在安装 Windows 9X 时，有安装“快速查看”（Windows 附带的一个工具）组件，可通过右击相应程序来查看其依赖的子系统。

图 3-8 是 Hello.exe 查看的结果。

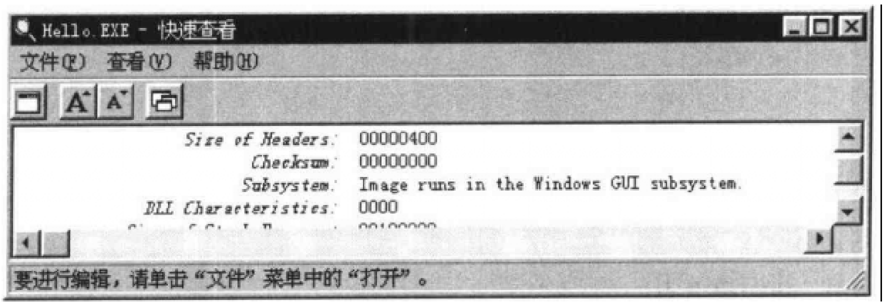


图 3-8

图 3-9 是 Console.exe 查看的结果。

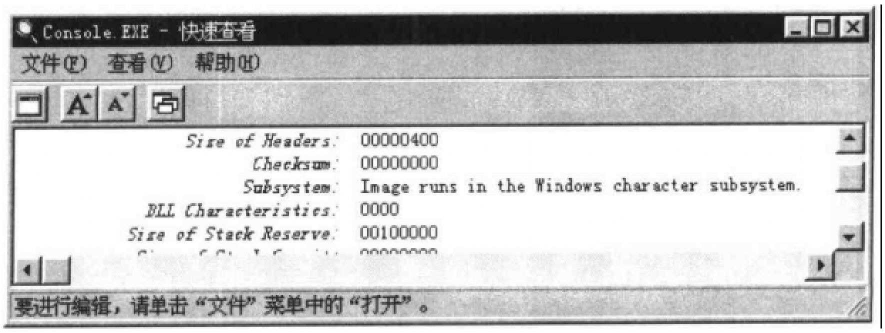


图 3-9

3.3 编写虚拟设备驱动程序

3.3.1 VxD 概述

虚拟设备驱动程序（Virtual x Driver 简称 VxD），从它的名字看来，它是硬件的驱动程序。“虚拟”是指它为 Ring 3 程序提供一个虚拟的硬件，从而使得各个 Ring 3 程序好象完全拥有该硬件，使得它好象独占一台机器（所谓“虚拟机”），而不需考虑其他 Ring 3 程序也在使用该硬件。

VxD 中的 x 意思是指各种设备，如 Windows 提供的键盘（keyboard）、鼠标（mouse）、定时器（timer）的驱动程序分别是 vkd、vmd、vtd 等。

例如，如果需要写一个声卡的驱动程序，其重要的目的之一是使得在同一时间里使用声卡发声的两个或几个 Ring 3 程序都正常工作，不相互干扰。如果它们不可能同时工作，则应该给出错误信息。

但 VxD 并不一定要驱动硬件。事实上，它本身也只是一个程序，有的 VxD 也可以象 DLL 一样，是可以动态加载的。但与 DLL 不同，VxD 是运行在 Ring 0 的特权级上的，它拥有系统的最高控制权。

VxD 只能运行在 Windows 9X 上（我们不讨论 16 位 Windows），它在 Windows NT/2000 上不能正常运行，Windows NT/2000 有它们自己的驱动程序模型。因此，如果用户的程序依赖于 VxD，则它失去了通用性，不能在 Windows NT/2000 上运行。

编写 VxD 需要的工具是 95DDK 或 98DDK，它们可以在 Microsoft 的网站上免费下载，本书的 VxD 都使用 98DDK。

另外还需要 Masm 6.11c 以上的版本，98DDK 本身也带有 Masm，版本是 6.11d，用户也可以去寻找一个新版本来使用。

另外，如果用户使用 C 来编写 VxD，那么，可能还需要 Visual C++，虽然也可以使用别的 C 编译器，但 Visual C++更适合一些。

本书作者使用的是 6.0 版的 VC。现在也出现一些使用 C++编写 VxD 的工具如 VtoolsD，但是，我们这里不讨论它们。

从加载的时间来分，VxD 可以分两种：动态的 VxD 和静态的 VxD。静态 VxD 必需在系统启动时加载，并且不能卸载。动态 VxD 在需要时由程序动态地加载/卸载。

静态 VxD 可以用实模式 DOS 下的 TSR 程序在 Windows 初始化时加载，我们不讨论这种方法；也可以通过在注册表或 system.ini 文件中写入 VxD 文件的名字来加载。动态 VxD 则在系统起动后由 Ring 3 程序加载。静态 VxD 和动态 VxD 的程序结构很相似，但它们各自要作一些不同的处理。

3.3.2 动态 VxD

1. 汇编语言版

下面介绍使用汇编语言写的一个动态 VxD 例子，它加载时显示如图 3-10 所示的对话框。源程序将在后面列出。这个程序只有三个文件，包括.asm 源文件、.def 定义文件和

makefile 文件，看上去虽然不短，但其实很多都是例行公事，真正要写的东西不多。

图 3-10

首先是源文件 AsmHello.asm。masm 的语法与 tasm 略有不同，但它并不难理解。使用 masm，其程序看起来更象 C 语言编写的程序，很多时候 masm 比 tasm 更好用。

源程序开始用 include 语句包含一些 DDK 提供的头文件。在这些头文件中有下面要用到的一些宏，如 BeginProc、EndProc、Declare_Virtual_Device、Begin_Control_Dispatch、End_Control_Dispatch 等的声明和一些函数的原型。

第一个用到的宏是 Declare_Virtual_Device，这是 VxD 的声明。这个宏生成一个称为设备描述块（Device Describe Block，DDB）的数据结构，在 Vmm.inc 里，它的名字是 VxD_Desc_Block，它包含有很多项，如 VxD 的名字、版本、DDK 版本等。DDB 必须在.def 文件中被导出以便系统获取 VxD 的信息。

Declare_Virtual_Device 它的用法如下：

Declare_Virtual_Device Name, MajorVer, MinorVer, CtrlProc, DeviceNum, InitOrder, V86Proc, PMProc

其中：

1）Name 是虚拟设备的名字，这个名字要全部用大写，并且这个名字在系统中应是惟一的。名字最多占 8 个字节。

2）MajorVer 和 MinorVer 分别指明 VxD 的主版本号和次版本号。

3）CtrlProc 是设备控制函数的名字。设备控制函数是用于分发系统消息处理时的相应处理程序，它们有些类似于普通 Win32 程序的消息处理函数。这个函数将在下面用 Begin_Control_Dispatch 宏和 End_Control_Dispatch 宏来生成。设备控制函数的名字一般是 VxD 的名字加上字串“_control”，在这里是 ASMHELLO_control。

4）DeviceNum 是标识 VxD 的惟一 ID，为避免与其他 VxD 冲突，可到 Microsoft 处取得一个 ID。但一般来说，可把它定义成 Undefined_Device_ID。但在下面的几种情况下，用户就需要一个 ID：

① 其他 VxD 要调用本 VxD 的服务。

② VxD 在初始化 int 2F 的 1607H 功能时，通知实模式程序它已经存在。

③ 实模式程序要调用 int 2F 的 1707H 功能加载该 VxD。

5）InitOrder 是初始化顺序。Windows 按这个顺序来加载各个 VxD。若用户的 VxD 要依赖于另一个 VxD，则这个 InitOrder 应设成比要依赖的 VxD 大。一般来说，这个值可以设成 Undefined_Init_Order。

6）V86Proc 和 PMProc 是虚拟 86 模式和保护模式 API 的处理函数。这是本 VxD 提供给其他程序使用的 API。这两个参数可以不填。

下面介绍 Begin_Control_Dispatch、End_Control_Dispatch 和 Control_Dispatch 宏。这三个用于生成一个消息分派函数，它根据得到的消息来调用指定的处理函数。使用方法如下：

```
Begin_Control_Dispatch Name
        Control_Dispatch Message, Procedure,CallType
        ......
End_Control_Dispatch Name
```

其中：

1）Name 是 VxD 的名字。

2）Message 是收到的消息。

3）Procedure 是对应的消息处理函数。

4）CallType 是函数调用方式，如 C 调用方式 cCAll，标准调用方式 sCall，pascal 调用方式 pCall。

宏 Control_Dispatch 也可写成：

```
Control_Dispatch    Message, Procedure, CallType, <Param1, Param2, ...>
```

其中“<......>”中的是函数的参数。

分派函数也可以用 BeginProc 和 EndProc 宏直接生成：

```
BeginProc Name_Control
    Control_Dispatch    Message, Procedure, CallType, <Param1, Param2, ...>
    ......
    clc
    ret
EndProc Name_Control
```

因为系统用标志位来判断调用成功与否，因此后面用 clc 来消除标志位。

注意这个函数要放在 VxD_LOCKED_CODE_SEG 段中。

在这里，消息分派函数处理了 3 个消息：Sys_Dynamic_Device_Init、Sys_Dynamic_Device_Exit 和 W32_DeviceIOControl。它们分别在 VxD 初始化，VxD 退出和收到时控制命令时被调用。

可以使用的消息共有 30 多个，如表 3-1 所示。

表 3-1 可使用的消息

消息	描述
Begin_PM_App	一个保护模式应用程序开始
Close_VM_Notify	一个虚拟机正被关闭
Create_Thread	一个新的线程被建立
Create_VM	一个新的虚拟机被建立
Critical_Reboot_Notify	系统正在重启（中断已关闭）
Debug_Query	请求虚拟设备驱动程序 Debug 接口
Destroy_Thread	一个线程被关闭

续表 3-1

消息	描述
Destroy_VM	一个虚拟机被关闭
Device_Init	虚拟设备驱动程序正被初始化（中断开放）
Device_Reboot_Notify	系统正在重启（中断开放）
End_PM_App	一个保护模式程序正结束
Init_Complete	所有虚拟设备已初始化完成
PNP_New_Devnode	新的设备结点
Power_Event	电源消息
Reboot_Processor	重启处理
Set_Device_Focus	虚拟设备取得
Sys_Critical_Exit	系统终止（中断关闭）
Sys_Critical_Init	系统被初始化（中断关闭）
Sys_VM_Init	系统虚拟机被建立
Sys_Dynamic_Device_Exit	动态 VxD 卸载
Sys_Dynamic_Device_Init	动态 VxD 加载
Sys_VM_Terminate	系统虚拟机被关闭
System_Exit	系统关闭（中断开放）
Terminate_Thread	线程被关闭
Thread_Init	线程已被始化
Thread_Not_Executeable	线程已停止执行
VM_Critical_Init	虚拟机已被建立（中断关闭）
VM_Init	虚拟机已被建立（中断开放）
VM_Not_Executable	虚拟机已停止执行
VM_Resume	虚拟机恢复执行
VM_Suspend	虚拟机暂停
VM_Terminate	虚拟机被关闭
W32_DeviceIoControl	Win32 设备 IO 控制

这些消息的参数在 vmm.h 中有描述，限于篇幅，这里不一一详述。另外 vmm.h 中还定义了一些消息，在这里没有列出。

现在介绍宏定义 VxD_PAGEABLE_DATA_SEG、VxD_PAGEABLE_CODE_SEG、VxD_PAGEABLE_DATA_ENDS 和 VxD_PAGEABLE_CODE_ENDS。这些宏用来定义段及其属性。它的意义是一目了然的。PAGEABLE 意为可分页的，即它可以被系统的虚拟内存管理机制交换到硬盘上去。

还有一些其他的段，如 VxD_LOCKED_CODE_SEG、VxD_LOCKED_DATA_SEG 表示不可交换出内存的数据和代码等。

最后是 BeginProc 和 EndProc 宏，这两个宏用于定义函数。比起 Proc 和 endp 宏，它有很多与 VxD 有关的参数选择，这里暂且不管它。

```
;*****************************
;文件：AsmHello.asm          *
;功能：动态 VxD 的例子汇编版 *
;*****************************
        .386p
        include C:\98DDK\inc\win98\vmm.inc              ;头文件
        include C:\98DDK\inc\win98\shell.inc
        include C:\98DDK\inc\win98\debug.inc

Declare_Virtual_Device                  \
                ASMHELLO,               \
                1, 0,       \
                ASMHELLO_control,  \
        Undefined_Device_ID, Undefined_Init_Order

;消息分派
Begin_Control_Dispatch ASMHELLO
    Control_Dispatch Sys_Dynamic_Device_Init, OnDynInitAsm, cCall
    Control_Dispatch Sys_Dynamic_Device_Exit, OnDynExitAsm, cCall
    Control_Dispatch W32_DeviceIOControl,     OnDynCtlAsm,  cCall
End_Control_Dispatch   ASMHELLO

;可分页数据段
VxD_PAGEABLE_DATA_SEG
    Caption   db "VxD Say Hello",0
    VxDInit db "VxD 动态加载——汇编程序版本",0
VxD_PAGEABLE_DATA_ENDS

;可分页代码段
VxD_PAGEABLE_CODE_SEG
BeginProc SayHello                          ;显示对话框
    mov ecx, OFFSET32 VxDInit
    VMMCall Get_sys_vm_handle
    mov eax,MB_OK+MB_ICONEXCLAMATION
    mov edi,OFFSET32 Caption
    xor esi,esi
    xor edx,edx
    VxDCall SHELL_Message
    ret
```

```
EndProc SayHello

BeginProc OnDynInitAsm                ;VxD 初始化
    call SayHello
    mov   eax,1
    ret
EndProc OnDynInitAsm

BeginProc OnDynExitAsm                ;VxD 退出
    mov eax,1
    ret
EndProc OnDynExitAsm

BeginProc OnDynCtlAsm          ;IO Control
    xor eax,eax
    ret
EndProc OnDynCtlAsm

VxD_PAGEABLE_CODE_ENDS
end
```

现在来看.def 文件。在大多数情况下.def 文件一般不需作什么变化，要变的只是第一行 VxD 的名字和最后一行 DDB 的名字，当新建一个工程时，.def 文件中只需改变这两行即可。

第一行的单词 DYNAMIC 表示这是一个动态的 VxD，若是静态 VxD，则要删去这个词。

下面的 SEGMENTS 部分描述 VxD 中的段属性。

但因为这里描述的很多段在本例的 VxD 中是没有的，因此在编译时会有一堆警告，我们大可不必管它。

asmhello.def 文件：

```
VXD ASMHELLO DYNAMIC

DESCRIPTION 'Hello VxD assemble version'

SEGMENTS
    _LPTEXT        CLASS 'LCODE'    PRELOAD NONDISCARDABLE
    _LTEXT         CLASS 'LCODE'    PRELOAD NONDISCARDABLE
    _LDATA         CLASS 'LCODE'    PRELOAD NONDISCARDABLE
    _TEXT          CLASS 'LCODE'    PRELOAD NONDISCARDABLE
    _DATA          CLASS 'LCODE'    PRELOAD NONDISCARDABLE
```

```
    CONST         CLASS 'LCODE'   PRELOAD NONDISCARDABLE
    _TLS          CLASS 'LCODE'   PRELOAD NONDISCARDABLE
    _BSS          CLASS 'LCODE'   PRELOAD NONDISCARDABLE
    _MSOTABLE     CLASS 'MCODE'   PRELOAD NONDISCARDABLE IOPL
    _MSGDATA      CLASS 'MCODE'   PRELOAD NONDISCARDABLE IOPL
    _IMSGTABLE    CLASS 'MCODE'   PRELOAD DISCARDABLE IOPL
    _IMSGDATA     CLASS 'MCODE'   PRELOAD DISCARDABLE IOPL
    _ITEXT        CLASS 'ICODE'   DISCARDABLE
    _IDATA        CLASS 'ICODE'   DISCARDABLE
    _PTEXT        CLASS 'PCODE'   NONDISCARDABLE
    _PDATA        CLASS 'PDATA'   NONDISCARDABLE SHARED
    _STEXT        CLASS 'SCODE'   RESIDENT
    _SDATA        CLASS 'SCODE'   RESIDENT
    _DBOSTART     CLASS   'DBOCODE'   PRELOAD   NONDISCARDABLE
CONFORMING
    _DBOCODE      CLASS   'DBOCODE'   PRELOAD   NONDISCARDABLE
CONFORMINO
    _DBODATA      CLASS   'DBOCODE'   PRELOAD   NONDISCARDABLE
CONFORMING
    _16ICODE      CLASS '16ICODE' PRELOAD DISCARDABLE
    _RCODE        CLASS 'RCODE'
  EXPORTS
    ASMHELLO_DDB @1
```

Makefile 也是各工程大致一样，只需改改名字就行。

要注意的是这里的 Makefile 是按 Microsoft 的 Makefile 管理工具 nmake.exe 的语法来写的，因此编译时要使用 nmake.exe。

Makefile 文件：

```
all: ASMHELLO.vxd

OBJECTS = ASMHELLO.Obj

.asm.obj:
    ml -coff -DBLD_COFF -DIS_32 -W2 -c -Cx -DMASM6 -Zd -DDEBUG $*.asm

ASMHELLO.vxd: $*.def $(OBJECTS)
    link  -machine:i386  -def:$*.def  -map:$*.map  -vxd  vxdwraps.clb  -nodefaultlib
$(OBJECTS)
```

2、C 语言版

现在介绍使用 C 编写 VxD 的方法。下面的这个例子和上一个例子完成的是同一件事。

加载时如图 3-11 所示的动态 VxD 例子——C 语言版。

图 3-11

使用 C 编写 VxD 也要写一个.asm 文件，但它相对简单得多，只要写一个消息分派函数即可。

如下的 Chello.asm，只有 20 多行。BeginProc 宏后的 LOCKED 表示这个函数所在的段被锁定在内存中，不可交换到硬盘上。

```
;**************************
;文件：CHello.ASM          *
;功能：动态 VxD 的 C 版本 *
;**************************
    .386p
    include C:\98DDK\inc\win98\vmm.inc
    include C:\98DDK\inc\win98\vdmad.inc
    include C:\98DDK\inc\win98\shell.inc
    include C:\98DDK\inc\win98\debug.inc

Declare_Virtual_Device      \
            CHELLO,         \
            1, 0,           \
            CHELLO_control, \
        Undefined_Device_ID, Undefined_Init_Order
;消息分派
BeginProc   CHELLO_control, LOCKED
    Control_Dispatch Sys_Dynamic_Device_Init, OnDynInit, sCall
    Control_Dispatch Sys_Dynamic_Device_Exit, OnDynExit, sCall
    Control_Dispatch W32_DeviceIOControl, OnDevIOCtl,sCall, <ecx, ebx, edx, esi>
    clc
    ret
EndProc  CHELLO_control

    end
```

现在，我们就可以使用 C 来编写消息处理的实现函数了。要注意，上面.asm 文件中说明了各消息处理函数是用标准调用方式（sCall）来调用，因此，下面的 C 函数也要用

标准调用的关键字（_stdcall）。

C 文件内容如下：

```
/***************************\
*文件：Func.c              *
*功能：动态 VxD 的 C 版本*
\***************************/
#include <basedef.h>
#include <vmm.h>
#include <vwin32.h>
#include <shell.h>

BOOL _stdcall OnDynInit()
{
    HVM hvm = Get_Sys_VM_Handle();
    SHELL_Message( hvm,
                MB_OK|MB_ICONEXCLAMATION,
                "VxD 动态加载——C 语言版本",
                "VxD Say Hello",
                NULL,NULL );
    return TRUE;
}

BOOL _stdcall OnDynExit()
{
    return TRUE;
}

DWORD _stdcall OnDevIOCtl(DWORD    dwService,
                          DWORD    dwDDB,
                          DWORD    hDevice,
                          PDIOCPARAMETERS lpDIOCParms)
{
    return 0;
}
```

这个工程的.def 文件和上一个没什么大的区别，不再列出。

Makefile 也大致相同，但增加了对 C 文件的编译，本书作者使用的是 Visual C++ 6.0 的编译器。因为这是在命令行中编译，用户在使用时需设好路径和各环境变量，请参看附录。

Makefile 文件：

```
all: CHello.vxd

OBJECTS = CHello.Obj Func.obj

.asm.obj:
    ml -coff -DBLD_COFF -DIS_32 -W2 -c -Cx -DMASM6 -Zd -DDEBUG $*.asm

.c.obj:
    cl -c -Gs -Zdpl -Od -D_X86_ -YX -W3 -DDEBLEVEL=1 -DBLD_COFF -DDEBUG -DIS_32 $*.c
.cpp.obj:
    cl -c -Gs -Zdpl -Odb2 -D_X86_ -YX -W3 -DDEBLEVEL=1 -DBLD_COFF -DDEBUG -DIS_32 $*.cpp
CHello.vxd: $*.def $(OBJECTS)
    link -machine:i386 -def:$*.def -map:$*.map -vxd vxdwraps.clb -nodefaultlib $(OBJECTS)
```

3. 测试动态 VxD

上面两个 VxD 编译通过后，要测试它则要写一个程序来加载它。加载 VxD 应使用 Windows API 函数 CreateFile。

下面的程序用 Visual C++ 6.0 编译通过，它是一个 Win32 控制台应用程序，编译后把 testvxd.exe 复制到 VxD 所在的目录下。

```
/****************************\
 *文件：testvxd.cpp          *
 *功能：测试 VxD              *
\****************************/
#include "stdafx.h"
#include "windows.h"
#include "stdio.h"
int main(int argc, char* argv[])
{
HANDLE hvxd;
char buff[1024];
if ( argc != 2 )
{
        printf("Usage:TestVxD vxdfilename\n");
        return 1;
    }
    else sprintf(buff,"\\\\.\\%s",argv[1]);         //处理命令行
    hvxd = CreateFile(buff, 0, 0, NULL, 0,         //载入 VxD
```

```
                    FILE_FLAG_DELETE_ON_CLOSE, NULL);
    if (hvxd == INVALID_HANDLE_VALUE)         //成功否？
            printf("Load VXD \"%s\" error !\n",argv[1]);
    else
    {
            printf("Load VXD OK !\n");
            CloseHandle(hvxd);                    //卸载 VxD
    }
        return 0;
}
```

在 DOS 窗口中打入：

F:\MBK\VXD\CHello>TESTVXD CHELLO.VXD

或：

F:\MBK\VXD\AsmHello>TESTVXD ASMHELLO.VXD

即可看到 VxD 加载运行的结果。

3.3.3 静态 VxD

编写静态 VxD 与编写动态 VxD 大同小异，主要是把.def 文件的第一句由“VXD VxD 名 DYNAMIC”改为“VXD VxD 名”，即把“DYNAMIC”去掉。

静态 VxD 与动态 VxD 所能接收到的消息并不完全相同，最明显的是收不到 Dynamic_Device_Init 和 Dynamic_Device_Exit 消息，也不一定要处理 W32_DeviceIOControl 消息。

下面的例子是一个静态 VxD，它处理 Create_VM 和 VM_Terminate 消息，这两个消息在一个虚拟机建立和关闭时发给 VxD。

因为 Windows 9X 为每个虚拟 86 方式建立一个虚拟机，因此，载入此 VxD 后，每当用户打开或关闭一个 DOS 窗口就会弹出如图 3-12 所示的消息框。

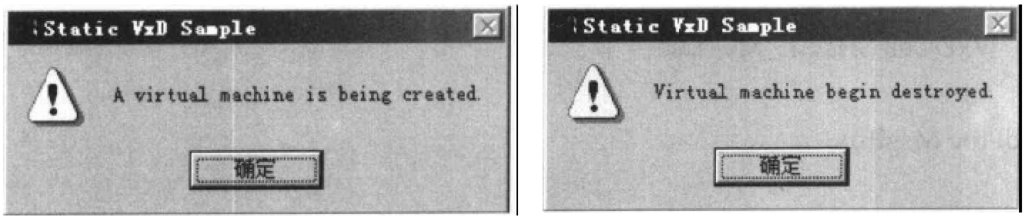


图 3-12

```
;*********************************
;文件：VMonitor.asm              *
;功能：静态 VxD 的例子            *
;      当 VM 建立和结束时通告     *
;*********************************
    .386p
```

```
include C:\98DDK\inc\win98\VMM.inc
include C:\98DDK\inc\win98\shell.inc

DECLARE_VIRTUAL_DEVICE        \
        VMONITOR,1,0,   \
        VMONITOR_Control,\
        UNDEFINED_DEVICE_ID, UNDEFINED_INIT_ORDER
;消息分派
Begin_control_dispatch VMONITOR
    Control_Dispatch Create_VM,         OnVMCreate
    Control_Dispatch VM_Terminate,      OnVMTerminate
End_control_dispatch VMONITOR

;可分页数据段
VxD_PAGEABLE_DATA_SEG
    szCaption       db "Static VxD Sample",0
    szVMCreate      db "A virtual machine is being created.",0
    szVMTerminate   db "Virtual machine begin destroyed.",0
VxD_PAGEABLE_DATA_ENDS

;可分页代码段
VxD_PAGEABLE_CODE_SEG
BeginProc MsgBox        ;显示对话框
    VMMCall Get_sys_vm_handle
    mov eax,MB_OK or MB_ICONEXCLAMATION
    mov edi, OFFSET32 szCaption
    xor  esi,esi
    xor  edx,edx
    VxDCall SHELL_Message
    ret
EndProc MsgBox

BeginProc OnVMCreate                ;虚拟机建立
        lea   ecx,szVMCreate
call  MsgBox
ret
EndProc OnVMCreate

BeginProc OnVMTerminate             ;虚拟机注销
lea   ecx,szVMTerminate
```

```
call MsgBox
ret
EndProc OnVMTerminate
VxD_PAGEABLE_CODE_ENDS
end
```

要加载静态 VxD，可以在实模式的 TSR 程序中加载，也可以通过写注册表或 System.ini 文件来加载。

写注册表的方法是在 HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\VxD 位置下建一主键，然后在主键下建一串值，取名为"StaticVxD"，其键值为 VxD 的路径和文件名。

写 System.ini 的方法是：编辑 Windows 目录下 System.ini 文件，找到[386enh]会话项，在其中加上一句"device=VxD 路径和文件名"即可。也可以不写路径，而把 VxD 文件复制到 Windows 的 System 目录下。

一般调试 VxD 时应该用编写 System.ini 文件的方法，因为万一 VxD 程序有什么问题导致 Windows 启动不了，可以在 DOS 下把 System.ini 中的这一句去掉，但在 DOS 下修改注册表则比较麻烦。

第四章 Windows 95/98/NT 内存管理

本章描述 Win32 下内存的管理调度、分配的情况；在 Windows 95/98/NT 下写时拷贝的异同，并给出例程；以及在 Windows 95 中读写绝对内存的方法。

4.1 Windows 95/NT 内存布局

内存管理是操作系统重要的基本功能之一。

我们知道，80X86 CPU 的分段管理功能使得程序运行时可以有多个段，每个段的长度最大可以是 4GB，而全局描述符表 GDT 和局部描述符表 LDT 里各可放入 8 千多个描述符，因此，80X86 CPU 的虚拟地址空间可达 64TB。但是，这种内存管理模式是十分麻烦的，程序员要在各个段之间像玩魔术一样变来变去，Win16 的程序员就是这样做的。现在，Windows 95/98/NT 的程序员再不须理会这些了，因为在 Win32 编程中已经没有段的概念了（至少对高级语言程序员来说是如此）。

Win32 的平坦内存模式使每个进程有 4GB 的内存空间，程序的代码和数据都放在同一地址空间中，即不必区分代码段和数据段。程序员当然也不需了解段寄存器 CS、DS、ES 等的具体内容。

如图 4-1 所示是 Windows 95 进程的地址空间。

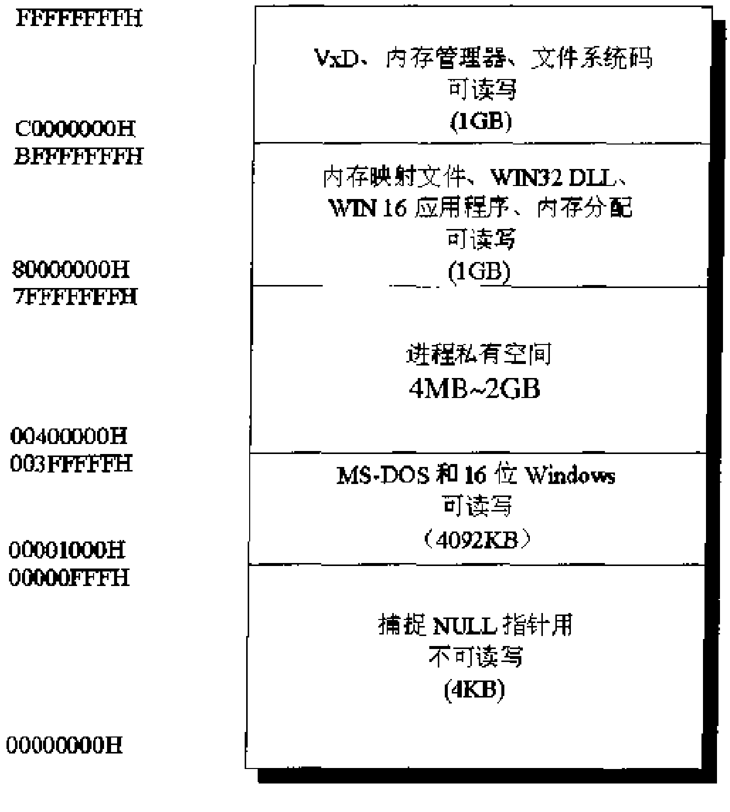


图 4-1

图 4-2 所示的是 Windows NT 进程的地址空间。

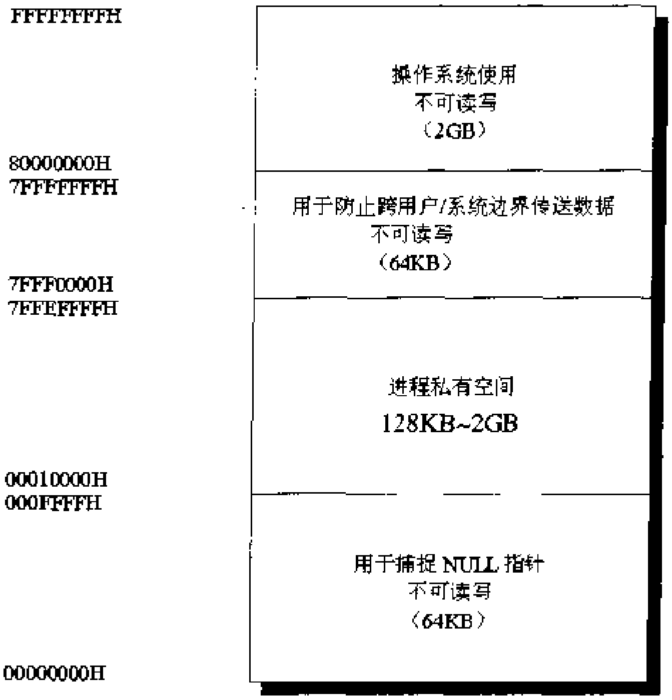


图 4-2

我们都知道，代码段是不可写的，而上面代码和数据却都放在一起，这其实是一种假像。在运行 Win32 进程的指令时，程序的段寄存器值是不相同的，但是，由于各段的基地址都是为 0，所以看起来它们像在同一地址空间中，但其实它们在不同的段中。

下面是在 Windows 98 中运行前面的 Hello.exe 时，用 Sofe-Ice 查看寄存器和段属性的结果。Windows 95 的情形跟 Windows 98 几乎完全一样，仅仅是段寄存器的值不同。

```
cpu

Processor 00 Registers
----------------------
CS:EIP=0167:00401002   SS:ESP=016F:0063FE38
EAX=00401000        EBX=00000000    ECX=815B1894    EDX=815B18D4
ESI=815B1874        EDI=00000000    EBP=0063FF78    EFL=00000A86
DS=016F   ES=016F   FS=0E17 GS=0000
……                 ；这里略去了一些输出
:ldt 167
Sel.    Type        Base        Limit           DPL     Attributes
0167    Code32      00000000    FFFFFFFF        3       P          RE
:ldt 16f
Sel.    Type    Base        Limit       DPL     Attributes
016F    Data32  00000000    0000AC00    3       P         RW ED
```

```
:ldt e17
Sel.    Type     Base       Limit       DPL    Attributes
0E17    Data16   815B1ACC   000000373   P      RW
```

GS 放的是空选择子，不必管它。

CS 的值是 0167H，用 ldt 命令查看 167H 在局部描述符表 LDT 中对应的段属性，可以看到它的基地址是 0，可读可执行（RE）；DS、ES、SS 的值为 016FH，同样用 ldt 命令查看，基地址是 0，可读可写（RW）。

FS 的内容是 0E17H，基地址却不是 0，这是因为 FS 段中装的并不是普通的用户数据，而是一个异常处理程序的链表，我们在后面章节中再详细讨论它。

Windows NT 的情形也与此相似：

```
:cpu

Processor 00 Registers
----------------------
CS:EIP=001B:00401002     SS:ESP=0023:0012FFC0
EAX=00000000       EBX=7FFDF000   ECX=00000201   EDX=FFFFFFFF
ESI=0012FA77       EDI=00000012   EBP=0012FFF0   EFL=00000216
DS=0023            ES=0023        FS=0038        GS=0000
......             ；这里略去了一些输出
:gdt 1b
Sel.   Type     Base       Limit      DPL   Attributes
001B   Code32   00000000   FFFFFFFF   3     P      RE
:gdt 23
Sel.   Type     Base       Limit      DPL   Attributes
0023   Data32   00000000   FFFFFFFF   3     P   RW
:gdt 38
Sel.   Type     Base       Limit      DPL   Attributes
003B   Data32   7FFDE000   00000FFF   3     P      RW
```

图 4-1 和图 4-2 中有些项标明了是否可读写，这仅仅是说明它客观上可否读写，并非说你就应该读写它们！这是因为 Windows 95/98 本身并不安全，它把一些本应保护起来的东西任由用户程序读写，而如果你这么做了，就可能会破坏操作系统，令 Windows 95/98 崩溃。

在标明“可写”的部分中，也并不是什么都可写的，例如，对于系统 API 函数的代码，在 Windows 95/98 中页属性为 Read only。以 MessageBoxA 为例，在 Sofe-Ice 中用 Page 命令查看页属性得如下结果：

```
:page messageboxa
Linear     Physical   Attributes   Type
BFF541BA   0054D1BA   P   AUR      Private
```

页属性是用户（U）、只读（R），因此它是不可写的。

有趣的是，进程空间最低的几 KB 是用于捕捉空指针的。当用户进程要分配内存时，可能会使用 HeapAlloc 之类的 API，返回值是一个指向所分配的内存的指针，但却有可能会因失败而返回 NULL。这时如果用这个空指针来存取数据，操作系统就可以捕捉到错误。例如下面的指令在 Windows 95/98/NT 都会引起异常：

```
xor esi,esi          ;ESI 置 0（NULL）
mov eax,[esi]        ;读地址 0
```

但是，Windows 9X 和 NT 对此的实现方法不同。

我们看看在 Windows 95/98 下的情况：

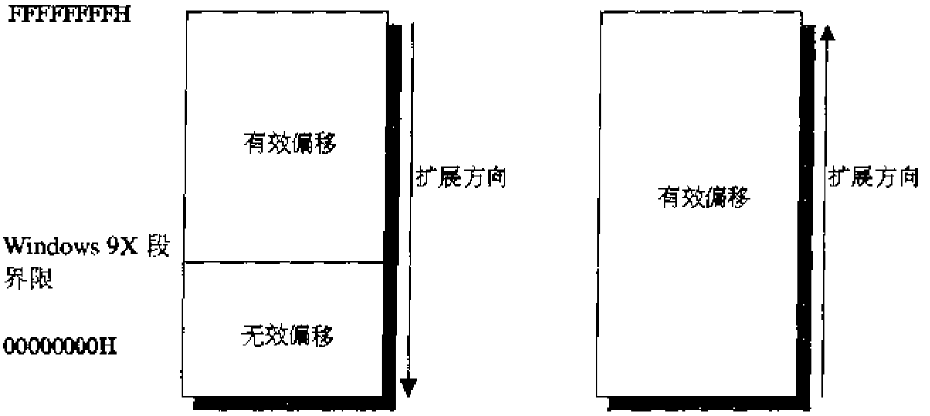
```
:page 0
Linear      Physical    Attributes    Type
00000000    00000000    P D A U RW    Hooked
```

Windows NT 下的情况：

```
:page 0
Linear      Physical    Attributes
00000000    NP 00000000
```

Windows NT 下线性地址 0 所在页的属性是不存在（NP），对地址 0 的读写当然会引起异常，但问题是 Windows 95/98 下该页却不但存在（P），而且还有可读可写（RW）的属性，那么异常是如何引起的呢？

答案是 Windows 95/98 并不是靠“页异常”来捕捉空指针操作，它依靠的是“段越界异常”来捕捉空指针操作。前面已介绍过用 ldt 或 gdt 指令查看 Windows 95/98 和 Windows NT 的 32 位进程数据段，它们是不同的。Windows NT 的数据段是普通数据段，段界限是 FFFFFFFFH，即 4GB；而 Windows 95/98 的数据段有 ED 标志，即它的扩展方向是向下扩展，而且，在上面例子中我们看到它的段界限是 AC00H。即在上例中，有效偏移量是 AC00H 到 FFFFFFFFH，用户读 0 地址的内容就会产生段越界异常。Windows 9X/NT 的段保护情况如图 4-3 的（a）、（b）所示。



（a）Windows 95/98　　（b）Windows NT

图 4-3

4.2 操纵内存

Win32 的内存 API 可以分为三类：虚拟内存管理、堆管理、内存映射文件管理。

1）虚拟内存管理则较适应于程序要使用大块内存的情况。

2）堆管理适用于程序要经常分配小块内存的情况。

3）内存映射文件则为大文件的操作提供方便，并提供进程间通讯的方法。

下面我们分开成三小节来讨论它们。

4.2.1 虚拟内存

虚拟内存适用于管理较大块的内存。一个例子是电子表格，你需要一大块连续的内存，但又并不是内存每一个都用到，这时应该怎么办？你当然可以直接申请这么一大块内存，但这样对宝贵的系统资源是十分浪费的；另一种方法是需要时动态分配，但这样得到的内存是不连续的，你为了记住它们之间的位置关系，必须用如链表或用一个二维指针数组等方法来维护它们的位置关系。

虚拟内存管理 API 则提供了另一种解决方法。因为运用虚拟内存管理 API 分配内存时，可以指定为保留内存，而不是立即分配内存。这样，当你以后真正要用内存时，就可以在保留的内存块中分配一块小的内存，而不需一下子提交许多不必要内存。

1. 分配/保留虚拟内存

```
mem dd ?
……
call VirtualAlloc,lpMem,Size,Type,Access
cmp eax,NULL
jz Error
mov [mem],eax
```

其中：

1）参数 lpMem 表示要分配/保留的内存地址，可以为 NULL。当要保留内存时，最好用 NULL，这样系统自己可以在适当的地方分配内存。

2）Size 表示要分配的内存的大小，单位是字节。

3）Type 是分配的类型，几个常用的是：

MEM_COMMIT 提交内存。

MEM_RESERVE 保留内存。

MEM_TOP_DOWN 在尽可能高的地址分配内存。这个标志只适用于 Windows NT。

4）Access 是保护标志。几个常用的是：

PAGE_READONLY 分配的内存是只读的。

PAGE_READWRITE 可读可写。

还有一些 PAGE_EXECUTE、PAGE_EXECUTE_READ 等，但我们知道，80X86 机器的页保护中只有读、写两项可设置，只要是可读的，就是可执行的。因些，这些属性对我们没有什么很大的意义。另外还有一些不常用的属性，不再细说。

2. 释放虚拟内存

call VirtualFree,lpMem,Size,Type

其中：

1）lpMem 表示要释放的内存的基地址。

2）Size，要释放的大小，单位为字节。

3）Type 释放的类型，可以是：

MEM_DECOMMIT，取消提交内存。

MEM_RELEASE，释放。

3. 改变页保护属性

call VirtaulProtect,lpMem,Size,Access,lpOldAccess

请 3 个参数的意义都有很显然，最后一个 lpOldAccess 是用来指向一个地址用以存放旧的页保护属性。若改变了多个页，则存放的是第一页的属性。

4. 内存锁定

call VirtualLock,lpMem,Size

用来确保当进程运行时，指定的内存总是在内存之中。

5. 内存解锁

call VirtualUnlock,lpMem,Size

是 VirtualLock 的反操作，解除锁定。

6. 实例

上面是常用的一些虚拟内存管理 API，有一点要注意的是，上面所有 API 的操作单位都是页（4096 字节），即是说，上页的参数 Size 单位虽然是字节，但 API 操作时会把它对齐到页。

下面是一个使用虚拟内存的简单例子，它先保留一块 3 页的内存，然后提交第 2 页，并做写入操作。

```
;**************************
;文件：Virtual.asm          *
;功能：使用虚拟内存          *
;**************************
.386p
.model flat,stdcall   ;平坦内存模式，标准调用

;常量定义
NULL   = 0
MB_OK = 0
MEM_RESERVE      =   2000H
MEM_COMMIT       =   1000H
MEM_DECOMMIT     =   4000H
MEM_RELEASE      =   8000H
```

```
MEM_FREE            =  10000H
MEM_PRIVATE         =  20000H
MEM_MAPPED          =  40000H
MEM_RESET           =  80000H
MEM_TOP_DOWN        =  100000H
MEM_IMAGE           =  1000000H

PAGE_NOACCESS       =  01H
PAGE_READONLY       =  02H
PAGE_READWRITE      =  04H
PAGE_WRITECOPY      =  08H
PAGE_EXECUTE        =  10H
PAGE_EXECUTE_READ               =  20H
PAGE_EXECUTE_READWRITE          =  40H
PAGE_EXECUTE_WRITECOPY          =  80H
PAGE_GUARD                      =  100H
PAGE_NOCACHE                    =  200H

;外部过程声明
EXTRN  VirtualAlloc        :PROC
EXTRN  VirtualFree         :PROC
EXTRN  VirtualLock         :PROC
EXTRN  VirtualProtect      :PROC
EXTRN  VirtualProtectEx    :PROC
EXTRN  VirtualQuery        :PROC
EXTRN  VirtualQueryEx      :PROC
EXTRN  VirtualUnlock       :PROC
EXTRN  MessageBoxA:        PROC
EXTRN  ExitProcess:        PROC

.data
     ErrCaption db "Error"
     ErrText   db "错误！"
     lpMem     dd ?
.code
main:
     ;保留
     call       VirtualAlloc,NULL,4096*3,MEM_RESERVE,PAGE_READWRITE
```

```
        mov     [lpMem],eax
        or      eax,eax
        jz      Error
        ;分配
        mov     ebx,[lpMem]
        add     ebx,4096
        call    VirtualAlloc,ebx,4096,MEM_COMMIT,PAGE_READWRITE
        or      eax,eax
        jz      Error
        mov     [eax],12345678H

        ;释放
        call    VirtualFree,[lpMem],NULL,MEM_RELEASE
        or      eax,eax
        jz      Error
        call    ExitProcess,0
Error:
        call    MessageBoxA,NULL,offset ErrText,offset ErrCaption,MB_OK
        call    ExitProcess,1
        end     main
```

4.2.2 堆管理

Win32 堆是进程的一块保留地址空间，它只属于进程（Win16 进程有全局堆和局部堆之分）。

1. 使用缺省堆

Win32 进程都有一个缺省的堆。该堆大小默认为 1MB，可以在编译时改变这个缺省值。Win32 的函数在运行时，若需要使用临时内存块则会从缺省堆中分配内存。由于有很多 Win32 函数使用缺省堆，因此 Windows 对缺省堆的使用进行了控制。

设想一个进程中有 2 个线程 A 和 B，它们同时要从堆中分配内存，其过程如下：

1）A 先运行，取得了缺省堆中的空闲地址。

2）在 A 尚未把地址置成已使用时，A 的 CPU 使用权被操作系统抢去，并被 B 取得控制权，这时 B 分配内存，它也取得了 A 相同的地址，并将其置为已使用。

3）这时又轮到 A 取得 CPU 使用权，A 不知道 B 已“夺取”了这块内存，以为它还是空闲的，因此，将其设置为自己“拥有”的内存，实际上它取得与 B 同一块内存。

以上过程就造成了 A 和 B 分配到同一地址的内存：这是一个相当严重的问题。因此，Win32 对堆的操作进行了序列化，即是在任一时刻只有一个线程在缺省堆上分配或释放内存，这在一定程度上会影响运行的速度。

取得缺省堆可通过下边的调用：

```
HeapHandle dd    ?
......
call GetProcessHeap
mov   [HeapHandle],eax
```

HeapHandle 中就存放了缺省堆的句柄。

并非一定要使用进程的缺省堆，也可以由用户自己创建堆来使用。这样做的一个好处是不用与 Win32 函数同用缺省堆而造成因序列化而降低运行速度。

2. 创建新的堆

创建一个堆可用下面的调用：

```
HeapHandle dd ?
......
call HeapCreate,flGption,dWINitialSize,cbMaximumSize
mov [HeapHandle],eax
```

其中 flOption 可以是：0、HEAP_NO_SERIALIZE、HEAP_GENERATE_EXCEPTIONS 或其组合。缺省为序列化操作，若指定了 HEAP_NO_SERIALIZE 则不会序列化堆操作。除非能保证没有 2 个或以上的线程同时对堆进行操作，否则不应使用这个标志。标志 HEAP_GENERATE_EXCEPTIONS 则表示进程在堆分配/再分配内存时，若失败则产生一个异常。

参数 dWINitalSize 和 cbMaximumSize 分别指出堆的初始化大小和最大容量。它的大小单位是 Byte，但 HeapeCreate 函数会把这个数字对向上齐到页的大小（4096 Byte）的整数倍，即，如果你指定的尺寸不足 1 页，它会给足 1 页。例如，如果是 4097Byte，则实际会给你 2 页，即 8182 字节。cbMaximumSize 可以为 0，Windows 会自己定一个值。

HeapCreate 成功时返回堆的句柄，否则返回 NULL。可用 GetLastError 函数取得错误码。

3. 分配堆内存

从堆中分配内存可以使用 HeapAlloc 函数：

```
pMem dd ?
......
call HeapAlloc,hHeap,dwFlags,dwBytes
mov [pMem],eax
```

其中：

hHeap 为堆句柄，由 HeapCreate 或 GetProcessHeap 取得。

dwFlags 是下列三个之一或其组合：HEAP_ZERO_MEMORY、HEAP_GENERATE_EXCEPTIONS 和 HEAP_NO_SERIALIZE。HEAP_ZERO_MEMORY 表示分配时把内存清 0。函数成功时返回所分配的指针，否则返回 NULL。

4. 重分配堆内存

可以改变所分配的堆内存的大小：

```
pMem1 dd ?
......
call HeapeReAlloc(hHeap,dwFlag,LpMem,dwBytes
mov [pMem1],eax
```

其中 lpMem 为 HeapAlloc 分配的堆内存；其他参数的含义跟上面几个一样，这里不再多说。

5. 释放堆内存

内存使用完后理应释放，语句如下：

```
call HeapFree,hHeap,dwFlags,lpMem
```

其中 dwFlags 为 0 或 HEAP_NO_SERIALIZE。函数失败则返回 0，成功则返回非 0 值。

6. 实例

下面是一个使用堆的例子：

```
;*****************************
;文件：;Heap.asm                *
;功能：演示堆的使用             *
;*****************************
.386p
.model flat,stdcall                ;平坦内存模式
extrn _wsprintfA:PROC
extrn GetProcessHeap:PROC
extrn HeapCreate:PROC
extrn HeapAlloc:PROC
extrn HeapReAlloc:PROC
extrn HeapSize:PROC
extrn HeapFree:PROC
extrn MessageBoxA:PROC
extrn ExitProcess:PROC

NULL   =0                          ;常量定义
MB_OK = 0                          ;
HEAP_NO_SERIALIZE                  equ 00000001H
HEAP_GROWABLE                      equ 00000002H
HEAP_GENERATE_EXCEPTIONS           equ 00000004H
HEAP_ZERO_MEMORY                   equ 00000008H
HEAP_REALLOC_IN_PLACE_ONLY         equ 00000010H

.data
```

```
        Caption db '堆使用例程',0                          (94)
        ErrText db '错误!',0
        Text db 400 dup (0)
        Param    db '缺省堆名柄:%X;',10,13
              db '新建堆句柄:%X' ,10,13
              db ' ________________________________',10,13
              db ' |分配块  |    原来|  重分配后    |',10,13
              db ' |-------|--------------|--------------|',10,13
              db ' | 大小   |%X      |%X       |',10,13
              db ' |-------|--------------|--------------|',10,13
              db ' | 地址   |%X      |%X       |',10,13
              db ' |-------|--------------|--------------|',10,13
              db 0
        DefHeap  dd ?

        MyHeap  dd ?
        Size1    dd ?
        Size2    dd ?
        lpMem1   dd ?
        lpMem2   dd ?

    .code
    main:
        call GetProcessHeap                              ;取缺省堆句柄
        mov [DefHeap],eax

        call HeapCreate,0,4096*3,0                       ;新建堆
        or    eax,eax
        jz    Error
        mov   [MyHeap],eax
        call HeapAlloc,eax,HEAP_ZERO_MEMORY,3000         ;分配内存块
        or    eax,eax
        jz    Error
        mov   [lpMem1],eax
        call HeapSize,[MyHeap],0,eax                     ;取得分配块的大小
        mov   [Size1],eax
        call  HeapReAlloc,[MyHeap],HEAP_ZERO_MEMORY,[lpMem1],8000       ;重分配
内存
```

```
or      eax,eax
jz      Error
mov     [lpMem2],eax
call HeapSize,[MyHeap],0,eax
mov     [Size2],eax
call HeapFree,[MyHeap],0,[lpMem2]          ;释放

call _wsprintfA,offset Text,offset, Param, DefHeap, MyHeap, Size1, Size2, lpMem1,
lpMem2
add         esp,4*8

call  MessageBoxA,NULL,offset Text,offset Caption,MB_OK
call  ExitProcess,0
Error:
call  MessageBoxA,NULL,offset ErrText,offset Caption,MB_OK
call  ExitProcess,0
end main
```

函数 HeapSize 作用是取得所分配堆内存的大小。

程序运行结果如图 4-4 所示。

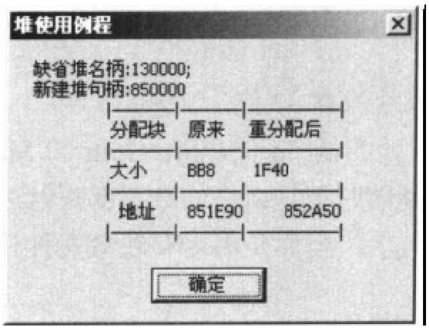


图 4-4

4.2.3 内存映射文件

内存映射文件是简化文件操作的一种途径。程序员使用内存映射文件，只要简单地将文件映射进内存，就可以通过读写内存来操纵文件的内容，而不需要频繁地打开文件、分配内存、读文件、操作文件内容，然后又写文件、关闭文件、释放内存。

1. 创建文件

创建内存映射文件要先用 CreateFile 来打开一个文件，这个函数在后面的章节会有详细讲述，这里简章介绍一下它的函数原型：

call CreateFileA, offset FileName, Access, ShareMode, offset Security, CreationDispositon, Flag, hTempFile

其中：

第一个参数是文件名。

第二个是如何程序要如何访问文件。

第三个是如何共享文件。

第四个是一个安全属性的指针。

函数成功则返回一个句柄，否则返回-1。

2. CreateFileMapping 函数

创建文件映象使用 CreateFileMapping 函数：

```
hMapping dd ?
……
call   CreateFileMapping hFile, offset Security ,Protect, SizeHigh,SizeLow,offset Name
mov [hMapping],eax
```

函数成功则返回一个句柄，否则返回 NULL，它的参数如下：

1）File 是 CreateFileA 所返回的句柄。

2）Security 是一个与安全有关的结构，要使用缺省值，可以传给函数参数 NULL。

3）Protect 是页面的保护属性，它可以是以下的其中之一：

① PAGE_READ：可读。

② PAGE_WRITE：可写。

③ PAGE_WRITECOPY：写时拷贝。

当要执行写操作时，必须在调用 CreateFileA 时给出 GENERIC_WRITE 和 GENERIC_READ 相“或”值。

4）SizeHigh 是文件最大尺寸的高 32 位。

5）SizeLow 是文件最大尺寸的低 32 位，SizeHigh 和 SizeLow 组成上个 64 位的值，即 180G 大小的文件。但你不能同时把这么大的数据都映射进内存。

6）Name 是这个映象的名字。当你要用内存映象文件进行进程间通迅时，这个名字就有用了。

3. 将数据映射入内存空间

将数据映射入内存空间的代码如下：

```
pMem   dd ?
……
call MapViewOfFile,hMapping, Access,OffsetHigh,OffsetLow,Count
mov [pMem],eax
```

函数成功则返回所映射的内存地址，失败则返回 NULL。函数的参数如下：

1）HMapping 是 CreateFileMapping 返回的名柄。

2）Access 是程序访问数据的方式，可以是以下的值：

① FILE_MAP_WRITE：可以读和写数据。

② FILE_MAP_READ：可以读。

③ FILE_MAP_ALL_ACCESS：可以读和写数据。与 FILE_MAP_WRITE 一样。

④ FILE_MAP_COPY：可以读和写数据。且写时拷贝。

3）OffsetHigh 所映射数据在文件中的偏移的高 32 位。

4）OffsetHigh 所映射数据在文件中的偏移的低 32 位。

5）Count：要映射的字节数。

4. 解除数据映射

解除数据映射的代码如下：

```
call UpmapViewOfFile,[pMem]
or eax,eax
jz Error
```

这个 API 的参数意义很明显，就是 MapViewOfFile 所返回的内存地址。成功则返回非 0，并会把数据写回进磁盘中；失败返回 0。

5. 刷新数据

刷新数据的代码如下：

```
call FlushViewOfFile,offset lpAddr,Count
or eax,eax
jz Error
```

这个 API 从指定的地址（lpAddr）开始，把指定数目（Count）写回文件。

6. 使用内存映射文件的实例

下面是一个使用内存映射文件的例子。

```
;*********************************
;文件：FileMap.asm             *
;功能：演示内存映射文件的使用    *
;*********************************
.386p
.model flat,stdcall
;外部过程声明
extrn CreateFileA:proc
extrn CreateFileMappingA:proc
extrn MapViewOfFile:proc
extrn UnmapViewOfFile:proc
extrn CloseHandle:proc
extrn ExitProcess:proc
extrn MessageBoxA:proc

;常量定义
NULL   = 0
MB_OK = 0
INVALID_HANDLE_VALUE = -1
```

```
GENERIC_READ        = 80000000H
GENERIC_WRITE       = 40000000H
FILE_SHARE_READ   = 1
FILE_SHARE_WRITE  = 2
FILE_SHARE_DELETE      = 4
CREATE_ALWAYS          = 2
PAGE_READWRITE         =  04H
FILE_ATTRIBUTE_NORMAL    = 00000080H
READ_CONTROL             = 00020000H
STANDARD_RIGHTS_REQUIRED   = 000F0000H
SECTION_QUERY                 = 0001H
SECTION_MAP_WRITE           = 0002H
SECTION_MAP_READ            = 0004H
SECTION_MAP_EXECUTE         = 0008H
SECTION_EXTEND_SIZE         = 0010H
SECTION_ALL_ACCESS          =    STANDARD_RIGHTS_REQUIRED    OR
SECTION_QU    ERYORSECTION_MAP_WRITEORSECTION_MAP_READOR    SECT
ION_MAP_EXECUTE OR SECTION_EXTEND_SIZE
FILE_MAP_COPY          = SECTION_QUERY
FILE_MAP_WRITE         = SECTION_MAP_WRITE
FILE_MAP_READ          = SECTION_MAP_READ
FILE_MAP_ALL_ACCESS  = SECTION_ALL_ACCESS

COUNT = 26
.data
    ErrCaption db '错误',0
    ErrText      db '发生错误',0

    FileName db 'Mapping.dat'
    hFile       dd ?
    hMapping dd ?
    pMapView dd ?

.code
main:
    call   CreateFileA,offset   FileName,GENERIC_WRITE   OR   GENERIC_READ,
FILE_SHARE_READ      OR      FILE_SHARE_WRITE,NULL,      CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL
```

```
        mov     [hFile],eax
        cmp     eax,INVALID_HANDLE_VALUE
        jz      Error
        call  CreateFileMappingA,[hFile],NULL,PAGE_READWRITE,0,COUNT,NULL
        mov     [hMapping],eax
        cmp     eax,NULL
        jz      Error
        call    MapViewOfFile,[hMapping],FILE_MAP_ALL_ACCESS,0,0,COUNT
        mov     [pMapView],eax
        cmp     eax,NULL
        jz      Error

        mov     ecx,COUNT                   ;写入内存映射文件
        mov     esi,eax
        mov     al,'A'
write:
        mov     byte ptr [esi],al
        inc     esi
        inc     al
        loop    write

        call  UnmapViewOfFile,[pMapView]    ;取消映射
        cmp     eax,0
        jz      Error
Exit:
        call  CloseHandle,[hMapping]        ;关闭文件及映射句柄
        call  CloseHandle,[hFile]
        call  ExitProcess,0
Error:
        call  MessageBoxA,NULL,offset ErrText,offset ErrCaption,MB_OK
        jmp   Exit
end main
```

程序在创建一个名为“Mapping.dat”的文件，并将其映射进内存，然后将“A”~“Z”26个字母写进内存，最后解除映射和关闭名柄。

作为一个简单的例子，这个程序并没有充分表现出内存影射文件的好处。但是，当你要处理的文件比较大时，就可以用 MapViewOfFile 把要处理的部分映射进内存，而把不需要的部分用 UnmapViewOfFile 写回文件。在这过程中不需要程序员做任何内存分配和释放的动作；还有一个好处是整个处理过程中，文件的当前指针和内存中的相对偏移地

址是相同的。

内存映射文件的另一个作用是用作进程间通讯。

当用 CreateFileMappingA 创建一个内存映射文件时，最后一个参数是一个名字，这个名字可以为 NULL。当你指定了一个名字，而下次如果又用这个名字调用 CreateFileMappingA，它们则会得到相同的映射，两个不同的进程就可以通过这块映射内存来进行数据交换。

一般说来，打开一个已存在的内存映射文件不用 CreateFileMappingA 函数，而应该用 OpenFileMappingA，例如：

```
hMapping dd ?
......
call OpenFileMappingA,Access,Inherit,offset MappingName
cmp   eax,NULL
jz    Error
mov   [hMapping],eax
```

其中：

第一个参数 Access 是访问的方法。

第二个参数 Inherit 决定是否可以被其他进程继承。

第三个参数 MappingName 就是用 CreateFileMappingA 建立的内存映射文件的名字。

7. 实例

下面的 2 个程序利用了内存映射文件来实现进程间的通讯。例程 1 的编辑控制的内容随例程 2 的编辑控制的内容变化而变化。

例程 1 应先运行，它在消息 WM_INITDIALOG 处理中建立一个名为“MyMappingName”的文件映射；然后建立一个定时器，并在 WM_TIMER 处理中调用 SetDlgItemTextA 函数，它的字串参数指向分配给内存映象文件的地址，这样，当内存映象文件有改变，例程 1 都能很快地在编辑控制中反映出来。

例程 1：

程序的源文件——First.asm：

```
;*********************************
;文件：First.ASM                 *
;功能：利用内存映射文件进行      *
;            进程间通讯之程序 1  *
;*********************************
IDC_EDITTEXT   = 100         ;EDIT 控制的 ID

.386p
.model flat,stdcall          ;平坦内存模式，标准调用约定
include win32.inc            ;包含有 Win32 部分常量定义的文件
extrn CreateFileA:proc
```

```
extrn CreateFileMappingA:proc
extrn OpenFileMappingA:proc
extrn MapViewOfFile:proc
extrn UnmapViewOfFile:proc
extrn CloseHandle:proc
extrn MessageBoxA:proc
extrn ExitProcess:proc
extrn GetModuleHandleA:proc
extrn DialogBoxParamA:proc
extrn EndDialog:Proc
extrn SetDlgItemTextA:proc
extrn SetTimer:proc
extrn KillTimer:proc

NULL   = 0
MB_OK = 0
INVALID_HANDLE_VALUE = -1
GENERIC_READ   = 80000000H
GENERIC_WRITE = 40000000H
FILE_SHARE_READ    = 1
FILE_SHARE_WRITE   = 2
FILE_SHARE_DELETE = 4
CREATE_ALWAYS      = 2
PAGE_READWRITE          =   04H
FILE_ATTRIBUTE_NORMAL   = 00000080H
READ_CONTROL               = 00020000H
STANDARD_RIGHTS_REQUIRED   = 000F0000H
SECTION_QUERY          = 0001H
SECTION_MAP_WRITE     = 0002H
SECTION_MAP_READ      = 0004H
SECTION_MAP_EXECUTE = 0008H
SECTION_EXTEND_SIZE = 0010H
SECTION_ALL_ACCESS          =     STANDARD_RIGHTS_REQUIRED     OR
SECTION_QUERY   OR   SECTION_MAP_WRITE   OR   SECTION_MAP_READ   OR
SECTION_MAP_EXECUTE OR SECTION_EXTEND_SIZE
FILE_MAP_COPY         = SECTION_QUERY
FILE_MAP_WRITE        = SECTION_MAP_WRITE
FILE_MAP_READ         = SECTION_MAP_READ
```

```
FILE_MAP_ALL_ACCESS = SECTION_ALL_ACCESS

COUNT = 200

.data
        DialogName   db 'MyDialog',0
        CaptionFail  db '失败！',0
        TextFail  db 'API:DialogBoxParamA 调用失败！',0
        TEXT         db '河汉清且浅，相去复几许。',0
        TextLen = $-TEXT
        ErrCaption db '错误',0
        ErrText      db '发生错误',0

        FileName db 'Mapping.dat'
        hFile        dd ?
        hMapping dd ?
        pMapView dd ?
        MappingName db 'MyMappingName',0

.code
;主过程
Start:
        call GetModuleHandleA,0          ;取得模块句柄，返回结果在 eax 中
        ;调用 DialogBoxParamA 显示对话框
        call DialogBoxParamA,eax,offset DialogName,0,offset DialogProc,0
        cmp eax,0                        ;成功否？
        jnz   Exit                       ;成功则退出
        ;显示出错信息
        call MessageBoxA,0,offset TextFail,offset CaptionFail,MB_OK
Exit:
        call ExitProcess,0               ;退出进程
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;对话框消息处理过程
DialogProc   proc uses ebx edi esi, hWnd:DWORD, wMsg:DWORD, wParam:DWORD,
lParam:DWORD
        cmp wMsg,WM_INITDIALOG
        jz WmInitDialog
        cmp wMsg,WM_CLOSE
```

```
        jz WmClose
        cmp wMsg,WM_COMMAND
        jz WmCommand
        cmp wMsg,WM_TIMER
        jz WmTimer
        jmp ExitProc
    WmInitDialog:                    ;对话框初始化处理
        call DlgInit,hWnd
        jmp ExitProc
    WmClose:                         ;用户按下了 Alt+F4 或单击右上角 X 按钮
        call DlgClose,hWnd
        jmp ExitProc
    WmTimer:
        call SetDlgItemTextA,hWnd,IDC_EDITTEXT,pMapView
        jmp ExitProc
    WmCommand:
        mov ebx,wParam
        cmp bx,IDOK                  ;是按下了“退出”键?
        jnz ExitProc
        call DlgClose,hWnd
        jmp ExitProc
    ExitProc:                        ;退出消息处理
        xor eax,eax
        ret
    DialogProc endp
    ;++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    DlgInit proc, hWnd:DWORD
        call CreateFileA,offset FileName,GENERIC_WRITE OR
GENERIC_READ,FILE_SHARE_READ OR
FILE_SHARE_WRITE,NULL,CREATE_ALWAYS,FILE_ATTRIBUTE_NORMAL,NULL
        mov  [hFile],eax
        cmp  eax,INVALID_HANDLE_VALUE
        jz   Error
        call CreateFileMappingA, [hFile], NULL, PAGE_READWRITE, 0, COUNT, offset
MappingName
        mov  [hMapping],eax
        cmp  eax,NULL
        jz   Error
```

```
        call MapViewOfFile,[hMapping],FILE_MAP_ALL_ACCESS,0,0,COUNT
        mov    [pMapView],eax
        cmp eax,NULL
        jz Error
        mov edi,eax
        mov esi,offset TEXT
        mov ecx,TextLen
        cld
        rep movsb
        call SetTimer,hWnd,12345,500,NULL          ;设置定时器
        ret
Error:
        call MessageBoxA,NULL,offset ErrText,offset ErrCaption,MB_OK
        call DlgClose
        ret
DlgInit endp
;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
DlgClose proc, hWnd:DWORD
        call KillTimer,hWnd,12345
        call UnmapViewOfFile,[pMapView]
        call CloseHandle,[hMapping]
        call CloseHandle,[hFile]
        call EndDialog,hWnd,1          ;关闭对话框
        ret
DlgClose endp
;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
end Start
```

运行程序，结果如图 4-5 所示。

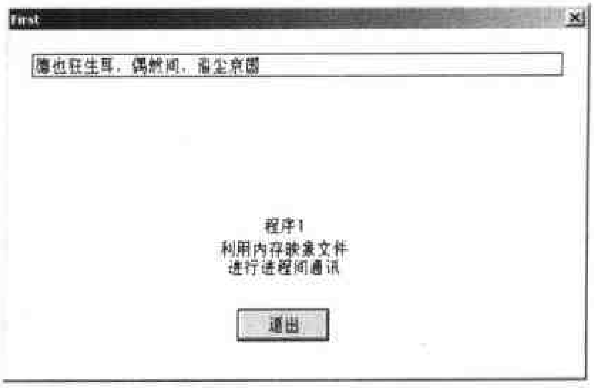


图 4-5

例程 2 在消息 WM_INITDIALOG 的处理中 OpenFileMappingA 打开例程 1 建立的文件映象。例程 2 也建立一个一个定时器，并在 WM_TIMER 处理中用 OetDlgItemTextA 来更新文件映象中的内容。

例程 2：

程序的源文件——Second.asm：

```
;****************************
;文件：Second.ASM            *
;功能：利用内存映射文件进行  *
;      进程间通讯之程序 2    *
;****************************
IDC_EDITTEXT   = 100    ;EDIT 控制的 ID

.386p
.model flat,stdcall        ;平坦内存模式，标准调用约定
include win32.inc          ;包含有 Win32 部分常量定义的文件
extrn CreateFileA:proc
extrn CreateFileMappingA:proc
extrn OpenFileMappingA:proc
extrn MapViewOfFile:proc
extrn UnmapViewOfFile:proc
extrn CloseHandle:proc
extrn MessageBoxA:proc
extrn ExitProcess:proc
extrn GetModuleHandleA:proc
extrn DialogBoxParamA:proc
extrn EndDialog:Proc
extrn SetDlgItemTextA:proc
extrn GetDlgItemTextA:proc
extrn SetTimer:proc
extrn KillTimer:proc

NULL   = 0
MB_OK = 0
INVALID_HANDLE_VALUE = -1
GENERIC_READ    = 80000000H
GENERIC_WRITE = 40000000H
FILE_SHARE_READ      = 1
FILE_SHARE_WRITE    = 2
```

```
    FILE_SHARE_DELETE = 4
    CREATE_ALWAYS       = 2
    PAGE_READWRITE            =  04H
    FILE_ATTRIBUTE_NORMAL   = 00000080H
    READ_CONTROL                 = 00020000H
    STANDARD_RIGHTS_REQUIRED   = 000F0000H
    SECTION_QUERY          = 0001H
    SECTION_MAP_WRITE     = 0002H
    SECTION_MAP_READ      = 0004H
    SECTION_MAP_EXECUTE = 0008H
    SECTION_EXTEND_SIZE = 0010H
    SECTION_ALL_ACCESS          =      STANDARD_RIGHTS_REQUIRED      OR
SECTION_QUERY   OR   SECTIGN_MAP_WRITE   OR   SECTION_MAP_READ   OR
SECTION_MAP_EXECUTE OR SECTION_EXTEND_SIZE
    FILE_MAP_COPY          = SECTION_QUERY
    FILE_MAP_WRITE         = SECTION_MAP_WRITE
    FILE_MAP_READ          = SECTION_MAP_READ
    FILE_MAP_ALL_ACCESS = SECTION_ALL_ACCESS

    COUNT = 200

    .data
        DialogName   db 'MyDialog',0
        CaptionFail  db '失败！',0
        TextFail     db 'API:DialogBoxParamA 调用失败！',0
        TEXT         db '盈盈一水间，脉脉不得语。',0
        TextLen = $-TEXT

        ErrCaption   db '错误',0
        ErrText      db '发生错误',0

        FileName db 'Mapping.dat'
        hFile        dd ?
        hMapping     dd ?
        pMapView     dd ?
        MappingName db 'MyMappingName',0

    .code
```

```
;主过程
Start:
    call GetModuleHandleA,0          ;取得模块句柄，返回结果在 eax 中
    ;调用 DialogBoxParamA 显示对话框
    call DialogBoxParamA,eax,offset DialogName,0,offset DialogProc,0
    cmp  eax,0                       ;成功否？
    jnz  Exit                        ;成功则退出
    ;显示出错信息
    call MessageBoxA,0,offset TextFail,offset CaptionFail,MB_OK
Exit:
    call ExitProcess,0               ;退出进程
;++++++++++++++++++++++++++++++++++++++++++++++++++++++
;对话框消息处理过程
DialogProc   proc uses ebx edi esi, hWnd:DWORD, wMsg:DWORD, wParam:DWORD,
lParam:DWORD
    cmp  wMsg,WM_INITDIALOG
    jz   WmInitDialog
    cmp  wMsg,WM_CLOSE
    jz   WmClose
    cmp  wMsg,WM_COMMAND
    jz   WmCommand
    cmp  wMsg,WM_TIMER
    jz   WmTimer
    jmp  ExitProc
WmInitDialog:                        ;对话框初始化处理
    call DlgInit,hWnd
    jmp  ExitProc
WmClose:                             ;用户按下了 Alt+F4 或单击右上角 X 按钮
    call DlgClose,hWnd
    jmp  ExitProc
WmTimer:
    call GetDlgItemTextA,hWnd,IDC_EDITTEXT,pMapView,COUNT
    jmp  ExitProc
WmCommand:
    mov  ebx,wParam
    cmp  bx,IDOK                     ;是按下了“退出”键？
    jnz  ExitProc
    call DlgClose,hWnd
```

```
        jmp    ExitProc

    ExitProc:                        ;退出消息处理
        xor    eax,eax
        ret
    DialogProc endp

    ;+++++++++++++++++++++++++++++++++++++++++++++++++++++
    DlgInit proc,hWnd:DWORD
        call OpenFileMappingA,FILE_MAP_ALL_ACCESS,0,offset MappingName
        mov    [hMapping],eax
        cmp    eax,NULL
        jz     Error
        call MapViewOfFile,[hMapping],FILE_MAP_ALL_ACCESS,0,0,COUNT
        mov    [pMapView],eax
        cmp    eax,NULL
        jz     Error

        mov    edi,eax
        mov    esi,offset TEXT
        mov    ecx,TextLen
        cld
        rep    movsb

        call SetDlgItemTextA,hWnd,IDC_EDITTEXT,pMapView
        call SetTimer,hWnd,12345,500,NULL
        ret

    Error:
        call MessageBoxA,NULL,offset ErrText,offset ErrCaption,MB_OK
        call DlgClose
        ret

    DlgInit endp
    ;+++++++++++++++++++++++++++++++++++++++++++++++++++++
    DlgClose proc,hWnd:DWORD
        call KillTimer,hWnd,12345
        call UnmapViewOfFile,[pMapView]
```

```
        call CloseHandle,[hMapping]

        call EndDialog,hWnd,1        ;关闭对话框
        ret
DlgClose endp
;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
end Start
```

运行程序，结果如图 4-6 所示。

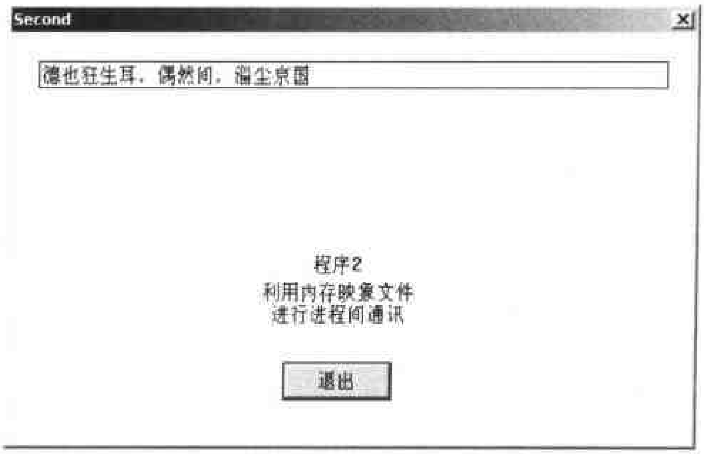


图 4-6

需要注意的是，虽然这两个进程取得的内存映射是一样的，但是，它们取得的内存虚拟地址未必是相等的。

在 Windows 9X 下，这两个进程的确是会取相同的虚拟内存地址，但是，若在 Windows NT 中运行，则并非如此，通常两个程序会得到不同的虚拟地址，但它们的内容会保持一致。

4.3 写时拷贝

写时拷贝（Copy On Write）是 Windows NT 所支持的一种内存优化技术，Windows 95 不支持它，但它也有一些类似于写时拷贝的行为。

写时拷贝并非 Windows 的创新，很多 Unix 系统都有使用这种技术。它的主要思想是：当两个或以上的进程在开始时有部分或全部相同的数据（例如执行同一程序的两个实例；又例如多个进程映射同一个 DLL 进自己的进程空间），这时，操作系统通过页映射把这些相同的页映射到相同的物理页，并标记为写时拷贝。然后，当其中某个进程要写这些页中的某一页时，操作系统在页异常处理过程中把该页复制一个副本，并将其映射到这个进程中，再执行写操作。这个操作由系统完成，一般不用程序员干预，但是，为了加强对 Windows 内存管理的理解，这里举了一个例子。

这个例程只能在 Windows 9X 下运行，它演示了 Windows 9X 下“写时拷贝”的过程。如图 4-7 所示，程序给出在用 WriteProcessMemory 写入前后的代码页的对应物理地址的

变化。

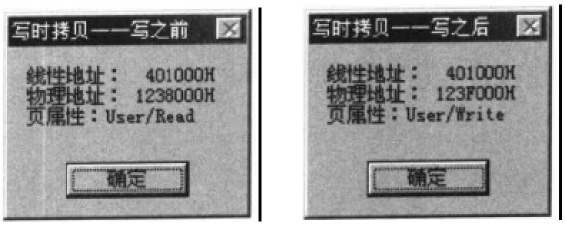


图 4-7

```
;*******************************************************
;文件：COW95.asm                                       *
;功能：演示 Windows 95/98 的写时拷贝(Copy On Write)功能 *
;*******************************************************
.386p
.model flat,stdcall
include win32.inc
extrn ExitProcess:PROC
extrn MessageBoxA:PROC
extrn _wsprintfA:PROC
extrn GetCurrentProcess:PROC
extrn WriteProcessMemory:PROC

CALL32 MACRO selector ,offsetv
        DB 09AH
        DD offsetv
        DW selector
       ENDM

.data
    LinearAddr    dd offset Start
    PhysAddrdd ?
    User      db 'User',0
    System        db 'System',0
    Read          db 'Read',0
    Write         db 'Write',0
    CaptionBefore db '写时拷贝——写之前',0
    CaptionAfter  db '写时拷贝——写之后',0
    Text      db 150 dup (0)
    ErrParam db '线性地址 %8XH 未映射入进程！',0
    SucParam      db '线性地址： %8XH',0dh,0ah
```

```
                db '物理地址：%8XH',0Dh,0Ah
                db '页属性：%s/%s',0
    Written     db ?

.code
Start:
    push offset L2P
    call  ToRing0Code
    push offset CaptionBefore
    call  Show
    Call        GetCurrentProcess
    call        WriteProcessMemory,eax,offset Modify,offset src,1,offset Written
    push        offset L2P
    call        ToRing0Code
    push        offset CaptionAfter
    call        Show
    call        ExitProcess,0

Show proc       Caption:DWORD           ;显示结果
    test        eax,1
    jnz         Present
    call        _wsprintfA,offset Text,offset ErrParam,LinearAddr
    add         esp,4*3
    call        MessageBoxA,0,offset Text,[Caption],MB_OK
    call        ExitProcess,1
Present:                        ;页存在
    mov         esi,offset User
    mov         edi,offset Read
    test        eax,100B
    jnz         Usr
    mov         esi,offset System
Usr:                            ;用户页
    test        eax,10B
    jz          R
    mov e       di,offset Write
R:      call    _wsprintfA,offset Text,offset SucParam,LinearAddr,[PhysAddr],esi,edi
    add         esp,4*6
    call        MessageBoxA,0,offset Text,[Caption],MB_OK
```

```
        ret
Show endp

ToRing0Code proc Ring0Proc:DWORD        ;生成调用门并调用 Ring0 子程序 Ring0Proc
        LOCAL Temp,Temp1
        call    GetLdtAddress           ;取出 LDT 的地址，返回结果在 eax 中
        mov     ecx,[eax]               ;保存 LDT 第一个描述符
        mov     [Temp],ecx          ;
        mov     ecx,[eax+4]         ;
        mov     [Temp1],ecx         ;
        mov edx,Ring0Proc               ;把调用门的内容写入 LDT
        mov [eax],dx                    ;偏移量的低 16 位
        mov word ptr [eax+2],28h        ;段选择子
        mov word ptr [eax+4],0ec00h     ;属性
        shr   edx,16                    ;偏移量的高 16 位
        mov [cax+6],dx          ;
        push eax
        CALL32  7,0             ;调用 Ring0 子程序
        pop  ebx
        mov edx,[Temp]          ;恢复 LDT 第一个描述符
        mov [ebx],edx      ;
        mov edx,[Temp1]         ;
        mov [ebx+4],edx         ;
        ret
ToRing0Code endp
GetLdtAddress proc              ;取 LDT 的地址
        push ebx                ;先要取 GDT 的地址
        sgdt [esp-2]            ;
        pop  ebx                ;
        sldt  ax                ;取 LDT 内容
        and   eax,0fff8H        ;屏蔽掉低 3 位、eax 的高 16 位清 0
        add   ebx,eax           ;算出 LDT 描述符的位置
        mov eax,[cbx+2]         ;从描述符中取出 LDT 的地址
        mov dl,[ebx+7]          ;
        shl   edx,24            ;
        and   eax,0ffffffh      ;
        or    eax,edx           ;
        ret
```

```
GetLdtAddress endp

L2P proc far                ;Ring0 程序，由线性地址取得物理地址及其页属性
    mov eax,cr3             ;CR3 是页表目录寄存器
    push eax                ;取页表目录的线性地址
    call  P2L               ;
    mov edx,LinearAddr      ;取出线性地址
    shr   edx,22            ;取高 10 位
    shl   edx,2             ;乘 4（每项 4 字节）
    mov eax,[eax+edx]       ;取对应页表的（物理）地址
    test  eax,1             ;存在否?
    jz    Quit
    and   eax,0fffff000H    ;清除属性位
    push eax                ;取对应页表的线性地址
    call  P2L               ;
    mov edx,LinearAddr      ;取线性地址第 12~21 位
    shl   edx,10            ;
    shr   edx,22            ;
    shl   edx,2             ;乘 4
    mov eax,[eax+edx]       ;取出物理页地址
    test  eax,1             ;存在否?
    jz    Quit
    push eax                ;保存 eax，用作返回值
    and   eax,0fffff000H    ;填入页内偏移
    mov edx,LinearAddr      ;
    and   edx,00000FFFH     ;
    or    eax,edx           ;
    mov [PhysAddr],eax      ;保存好物理地址
    pop   eax               ;返回对应的物理页和属性
Quit:
    ret
L2P endp

P2L proc ,Physcial          ;由物理地址得到线性地址
    push Physcial
    int   20H               ;VxDCall _MapPhysToLinear
    dd    0001006cH
    add   esp,4             ;调整堆栈
```

```
        ret
P2L endp

Modify      dd ?
src         dd 'AAAA'
End Start
```

4.4 在 Windows 95 里读写物理内存

在 Win32 中，进程看到的地址是线性地址，它已经过了分页机制的转换，看到的不再是物理地址。但是，有时需要对物理地址进行访问，例如，当要进行 DMA 编程时就必须这么做。

对物理地址进行访问方法是：调用 VMM 的函数_MapPhysToLinear，把物理地址映射到线性地址，然后再写线性地址，这个过程要用到 VxD 调用。最常见的调用 VxD 方式是写一个 VxD，但是，我们需要的是在普通的应用程序中能访问物理内存。若在 Win16 应用程序中，可以通过 DPMI 的调用实现，而在 Win32 中则要另想办法。

VxD 代码是 Ring0 代码，而 Win32 应用程序却是运行在 Ring3 中，因此不能直接用 Call 指令来调用 VxD 服务。Windows 95/98 提供的 WIN32 API 函数 DeviceIOControl 可以完成调用 VxD 的任务。DeviceIOControl 调用 C 语言的原型如下：

BOOL DeviceIOControl(HANDLE hDevice, DWORD dwIOControlCode, LPVOID lpinBuffer, DWORD cbInBuffer, LPVOID lpvOutBuffer, DWORD cbOntBuffer, LPDWORD lpcbBytesReturned, LPOVERLAPPED lpOverlapped);

这个函数用 CreateFile 打开的 VxD 句柄向 VxD 中指定的函数传送参数，并调用它，得到结果。本节不使用这种方法，而是通过生成调用门的方法实现。

我们知道在 80X86 系统中，Ring3 的代码要调用 Ring0 的代码，需要通过"门"。"门"有两种：中断门和调用门：

1）中断门位于中断门描述符表（IDT）中。

2）调用门可以位于全局描述符表（GDT）或局部描述符表（LDT）中。

一个健壮的操作系统应该对这几个系统表格进行隔离保护（Windows NT 这样做了），让用户程序不能随意访问它们。但不幸的是（或者应该说"幸运的是"）Windows 95/98 却没有这么做，由此，我们可以在以上的某个表格上生成一个中断门或调用门，使它指向我们的 Ring0 处理程序，这样我们的程序就可以由 Ring3 进入 Ring0，就可以直接调用 VxD 的函数了。

我们下面给出的程序是在 LDT 里生成一个调用门。采用 LDT 而不是 GDT 或 IDT 的理由是：LDT 开头几个描述符是空的，Windows 95 和 Windows 98 都是如此（Windows NT 的进程没有 LDT，即使它有，它也一定会保护起来，不让用户去动它）。我们来看看在 Windows 98 中用 Sofe-ice 查看的结果：

:LDT

Sel.	Type	Base	Limit	DPL	Attributes
LDTbase=80003000		Limit=3FFF			
0004	Reserved	00000000	00000000	0	NP
000C	Reserved	00000000	00000000	0	NP
0014	Reserved	00000000	00000000	0	NP
001C	Reserved	00000000	00000000	0	NP
0024	Reserved	00000000	00000000	0	NP
002C	Reserved	00000000	00000000	0	NP
0034	Reserved	00000000	00000000	0	NP
003C	Reserved	00000000	00000000	0	NP
0044	Reserved	00000000	00000000	0	NP
004C	Reserved	00000000	00000000	0	NP
0054	Reserved	00000000	00000000	0	NP
005C	Reserved	00000000	00000000	0	NP
0064	Reserved	00000000	00000000	0	NP
006C	Reserved	00000000	00000000	0	NP
0074	Reserved	00000000	00000000	0	P
007C	Reserved	00000000	00000000	0	NP
0087	Data16	80002000	00000FFF	3	P RW
……	；略去一些输出				

其实就算是用 GDT 或 IDT，只要注意保存和恢复好原来的内容，一般也不会出错。不过，还是小心操作为好。

程序将把调用门放在第 0 个描述符的位置上。描述符属性字中的描述符特权级（DPL）设置为 3，以便 Ring3 的进程能调用它；存在位 P=1；类型为 386 调用门即 TYPE 为 C；不需复制堆栈参数。其属性字会为 EC00H，如图 4-8 所示。

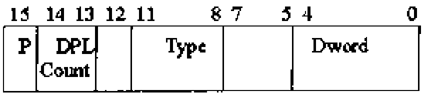


图 4-8

调用门中的段选择子又该是什么呢？我们看看 Sofe-Ice 的 gdt 指令输出：

:gdt

Sel.	Type	Base	Limit	DPL	Attributes
GDTbase=C0FDE000		Limit=0FFF			
……	；略去一些输出				
0028	Code32	00000000	FFFFFFFF	0	P RE
0030	Data32	00000000	FFFFFFFF	0	P RW
……	；略去一些输出				

GDT 中有 DPL 为 0 且基地址也为 0 的代码段和数据段，代码段选择子为 0028H，这

在 Windows 95 和 Windows 98 中是一样的，调用门如图 4-9 所示。

63 49	48 32	31 16	15 0
偏移量（31-16）	属性 = 0EC00H	段选择子 = 0028H	偏移量（15-0）

图 4-9

当 Ring3 进程 call 这个调用门时，选择子中的请求特权级也应为 3，因此，选择子的值为 0007H，如图 4-10 所示。

15	2	1 0
	0	1 1 1

图 4-10

在 Ring0 调用 VxD 采用的是 int 20 指令。当程序第一次调用这个 VxD 函数时，程序中放的是 Int 20 指令，但是，当 int 20 处理函数在执行时会改写这个调用处的代码，用一个 call 指令代替掉原来的 int 20 指令。下面是调用_MapPhysToLinear 前后代码的变化情况：

1）调用_MapPhysToLinear 前。

```
0028:004011BA    FF7508      PUSH    DWORD PTR [EBP+08]
0028:004011BD    CD20        INT     20 VXDCall _MapPhysToLinear
0028:004011C3    83C404      ADD     ESP,04
```

2）调用_MapPhysToLinear 后。

```
0028:004011BA    FF7508           PUSH    DWORD PTR [EBP+08]
0028:004011BD    FF15180E01C0     CALL    [_MapPhysToLinear]
0028:004011C3    83C404           ADD     ESP,04
```

下面是写指定物理内存的源程序。程序要往屏幕处于文本方式时的显存：地址 B8000H 写一串字符。本程序首先生成一个调用门，进入 Ring0，调用 VMM 的函数_MapPhysToLinear 取得物理地址对应的线性地址，然后传送数据到线性地址。

```
;*********************************************************************
;文件：rw.asm                                                          *
;功能：在 Windows 95/98 中访问物理内存                                    *
;在 Windows 95/98 里打开 DOS 窗口，用 ALT-ENTER 热键切换到全屏幕，        *
;在命令行方式下运行此程序。                                              *
;也可在 SOFT-ICE FOR Windows 95/98 的全屏方式下按 F8 单步跟踪看结果。 *
;*********************************************************************

CALL32 MACRO selector ,offsetv
        DB 09AH
        DD offsetv
        DW selector
       ENDM
```

```
AddressToWrite   equ 0B8000H         ;要写的地址

.386p
.model flat,stdcall
include WIN32.inc
extrn ExitProcess:PROC

.data
    Data      dw 10*80 dup (0e41H)      ;要写的数据
    DataLeng = $-Data
.code
Start:
    push offset MyProc                ;切换到 Ring0 并调用 MyProc
    call ToRing0Code                  ;
    call ExitProcess,0                ;退出进程

ToRing0Code proc Ring0Proc:DWORD      ;生成调用门，进入 Ring0 处理程序
Ring0Proc
    LOCAL Temp,Temp1
    call GetLdtAddress                ;取出 LDT 的地址，返回结果在 eax 中
    mov   ecx,[eax]                   ;保存 LDT 第一个描述符
    mov   [Temp],ecx                  ;
    mov   ecx,[eax+4]                 ;
    mov   [Temp1],ecx                 ;
    mov   edx,Ring0Proc               ;把调用门的内容写入 LDT
    mov   [eax],dx                    ;偏移量的低 16 位
    mov   word ptr [eax+2],28h        ;段选择子
    mov   word ptr [eax+4],0ec00h     ;属性
    shr   edx,16                      ;偏移量的高 16 位
    mov   [eax+6],dx                  ;
    push eax
    CALL32 7,0                        ;调用 Ring0 子程序
    pop   ebx
    mov   edx,[Temp]                  ;恢复 LDT 第一个描述符
    mov   [ebx],edx                   ;
    mov   edx,[Temp1]                 ;
    mov   [ebx+4],edx                 ;
    ret
```

```
ToRing0Code endp
GetLdtAddress proc              ;取 LDT 的地址
        push ebx                ;先要取 GDT 的地址
        sgdt [esp-2]
        pop   ebx
        sldt ax                 ;取 LDT 内容
        and   eax,0fff8H        ;屏蔽掉低 3 位、eax 的高 16 位清 0
        add   ebx,eax           ;算出 LDT 描述符的位置
        mov   eax,[ebx+2]       ;从描述符中取出 LDT 的地址
        mov   dl,[ebx+7]
        shl   edx,24
        and   eax,0ffffffh
        or    eax,edx
        ret
GetLdtAddress endp

MyProc:                         ;Ring0 处理程序，写指定的物理内存
        push AddressToWrite
        int   20H               ;VxDCall _MapPhysToLinear
        dd    0001006cH         ;
        add   esp,4             ;调整堆栈指针

        cli
        push ds
        push es
        mov   bx,30H            ;设定数据段
        mov   ds,bx
        mov   es,bx
        cld
        mov   ecx,DataLeng/4    ;传送数据
        mov   esi,offset Data
        mov   edi,eax
        rep   movsd
        pop   es
        pop   ds
        sti
        RETF                    ;远程返回
end Start
```

你也许会想到另一种极端的一些方法：暂时禁止分页。这虽然是一种更加危险（至少在我看来是如此）的做法，但是，它也是可行的。

这种方法有一点麻烦的是：当禁止分页和重新启用分页时，代码所在的页必须要物理地址等于线性地址，否则一旦启动或禁止分页机制，则 CPU 会弄错下一条要执行的指令的所在地址。幸运的是，Windows 95/98 中有一个事实，那就是它们的线性地址所在的页 0 就是物理地址 0 所在的页。我们要做的是把禁止/启用分页的代码、要写的数据和传送数据用的代码先送到地址 0 处执行。当然，还要保存和恢复好 0 页的内容。

首先在.data 处另上一句：

```
Temp0    db 4096 dup (?)      ;保存 0 页的内容
```

然后，把 MyProc 子程序换成以下的子程序：

```
MyProc:                          ;Ring0 处理程序，写指定的物理内存
     cli
     push ds
     push es
     mov   bx,30H                ;设定数据段
     mov   ds,bx                 ;
     mov   es,bx                 ;
     cld
     mov   ecx,4096/4            ;保存 0 页的内容
     mov   edi,offset Temp0
     mov   esi,0
     rep   movsd
     mov   ecx,To0Length/4+1     ;送代码到 0 页
     mov   esi,offset To0Code
     mov   edi,0
     rep   movsd
     mov   ecx,DataLeng/4+1      ;送数据到 0 页
     mov   esi,offset Data
     rep   movsd
     mov   eax,0                 ;跳转到地址 0 处执行
     jmp   eax               ;
SendOK:
     mov   ecx,4096/4            ;恢复 0 页内容
     mov   esi,offset Temp0
     mov   edi,0
     rep   movsd
     pop   es
     pop   ds
```

```
        sti
        RETF                        ;远程返回

ToOCode:                            ;这些代码将送往地址 0 处执行
        mov   eax,cr0               ;禁止分页
        and   eax,07FFFFFFFH
        mov   cr0,eax
        jmp   short DisP
DisP:
        mov   ecx,DataLeng/4        ;送数据到指定的物理地址
        mov   esi,(ToOLength/4+1)*4
        mov   edi,AddressToWrite
        rep   movsd
        or    eax,080000000H        ;Enable 分页机制
        mov   cr0,eax
        jmp   short EP
EP:     mov   eax,offset SendOK     ;跳回 SendOK 处执行
        jmp   eax
ToOLength=$-ToOCode
```

这种方法的一个问题是：若要传送的数据很多（例如，多于 1 页），则比较麻烦。

在分页机制作用之下，32 位的线性地址被分成三部分：最低的 12 位、中间 10 位、高 10 位，分别对应页映射机制的页表目录、页表、页内偏移。当 CPU 接到一个存取某一线性地址的内容的请求时，它把线性地址的这三部分分别在分页机制下对应进行部分查找，得到相应的物理地址和页属性，同时检查访问是否允许。我们也可以在汇编语言一级上模仿这个过程，求得线性地址对应的物理地址以及其页属性。

图 4-11 是程序运行的结果。

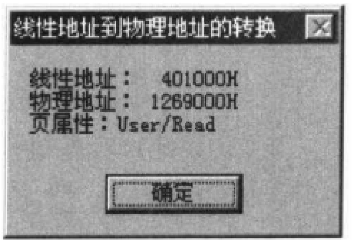


图 4-11

```
;*******************************************************
;文件：l2p.asm                                         *
;功能：通过查页表由线性地址得到相应的                  *
;          物理地址，并显示其页属性                    *
;*******************************************************
```

```
CALL32 MACRO selector ,offsetv
        DB 09AH
        DD offsetv
        DW selector
       ENDM

LinearAddr    equ offset Start       ;要查看的线性地址
.386p
.model flat,stdcall
include WIN32.inc
extrn ExitProcess:PROC
extrn MessageBoxA:PROC
extrn _wsprintfA:PROC

.data
    PhysAddr dd ?                ;存放得到的物理地址
    User    db 'User',0
    System      db 'System',0
    Read        db 'Read',0
    Write       db 'Write',0
    Caption   db '线性地址到物理地址的转换',0
    Text     db 150 dup (0)
    ErrParam db '线性地址  %8XH  不在内存中！ ',0
    SucParam db '线性地址：  %8XH',0dh,0ah
            db '物理地址：  %8XH',0Dh,0Ah
            db '页属性：  %s/%s',0
.code
Start:
    call ToRing0Code,offset L2P      ;取物理地址及页信息
    call Show                        ;显示物理地址及页信息
    call ExitProcess,0               ;退出进程

Show Proc
    test eax,1                       ;存在否？
    jnz   Present
    call _wsprintfA,offset Text,offset ErrParam,LinearAddr
    add    esp,4*3
    call MessageBoxA,0,offset Text,offset Caption,MB_OK
```

```
        call ExitProcess,1
Present:
        mov   esi,offset User
        mov   edi,offset Read
        test eax,100B                   ;用户页还是系统页？
        jnz   Usr
        mov   esi,offset System
Usr:test eax,10B                        ;只读还是可写？
        jz    RD
        mov   edi,offset Write
RD: call _wsprintfA,offset Text,offset SucParam,LinearAddr,[PhysAddr],esi,edi
        add   esp,4*6
        call MessageBoxA,0,offset Text,offset Caption,MB_OK
        ret
Show endp

ToRing0Code proc Ring0Proc:DWORD       ;生成调用门并调用 Ring0 子程序
;Ring0Proc
......                              ;略，请参看上面的例子
ToRing0Code endp
GetLdtAddress proc                  ;取 LDT 的地址
......                              ;略，请参看上面的例子
GetLdtAddress endp

L2P proc far                        ;由线性地址取得物理地址及其页属性
        mov   eax,cr3                    ;CR3 是页表目录寄存器
        push eax                        ;取页表目录的线性地址
        call P2L                        ;
        mov   edx,LinearAddr            ;取出线性地址
        shr   edx,22                    ;取高 10 位
        shl   edx,2                     ;乘 4（每项 4 字节）
        mov   eax,[eax+edx]             ;取对应页表的(物理)地址
        test eax,1                      ;存在否？
        jz    Quit
        and   eax,0fffff000H            ;清除属性位
        push eax                        ;取对应页表的线性地址
        call P2L                        ;
        mov   edx,LinearAddr            ;取线性地址第 12~21 位
```

```
        shl   edx,10                ;
        shr   edx,22                ;
        shl   edx,2                 ;乘 4
        mov   eax,[eax+edx]         ;取出物理页地址
        test eax,1                  ;存在否?
        jz    Quit
        push eax                    ;保存 eax，用作返回值
        and   eax,0fffff000H        ;填入页内偏移
        mov   edx,LinearAddr        ;
        and   edx,00000FFFH         ;
        or    eax,edx               ;
        mov   [PhysAddr],eax        ;保存好物理地址
        pop   eax                   ;返回对应的物理页和属性
Quit:
        ret
L2P endp

P2L proc ,Physcial                  ;由物理地址得到线性地址
        push Physcial
        int   20H                   ;VxDCall _MapPhysToLinear
        dd    0001006cH
        add   esp,4                 ;调整堆栈
        ret
P2L endp
END Start
```

第五章 进程和线程

本章主要介绍 Windows 的进程和线程及其编程，包括 Win32 进程，WinMain 函数的入口，建立线程和进程，线程调度和使用，写入远程进程，钩子过程等。

5.1 Win32 进程

Windows 是一个多任务系统，所谓的“多任务”，是指同一时间，可以有多个程序在内存中运行，称为进程。这种概念不同于 DOS 下的 TSR 程序，TSR 程序只是驻留在内存中，并不是总能够时时取得使用 CPU 的权力。如图 5-1 所示是 Windows NT 的“任务管理器”，它不但显示系统中有多个进程在运行，还显示进程之间在不停地切换。

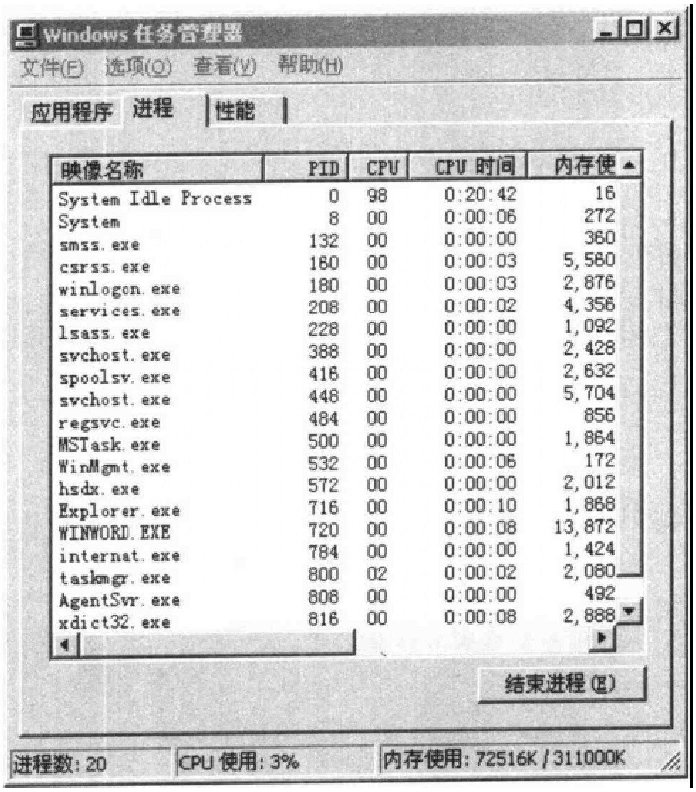


图 5-1

不同于 Unix 系统，在 Win32 系统中，进程是一个静态的概念，而线程才是一个动态的概念。

进程创建时，它就有了一个主线程，线程自己又可以生成线程。后面我们会讲到线程的创建和调度。

进程包含有环境变量，可以用下面的调用取得：

```
lpEnv    dd ?
    ......
call GetEnvironmentStrings
```

```
mov [lpEnv],eax
```

进程的命令行参数可通过下面的调用取得：

```
lpCL     dd ?
……
call GetCommandLine
mov   [lpCL],eax
```

前面我们说过进程之间被相互隔离，不能互相看到，但很多时候我们会希望我们的程序在某一时刻只有一个拷贝在运行，这时可用互斥函数 CreateMutex。它创建一个互斥对象，当它要创建的对象已存在时，GetLastError 函数会返回 ERROR_ALREADY_EXISTS，从而使我们知道是否已有一个本程序的拷贝在内存中。

```
;*********************************
;Mutex.asm                       *
;功能：使用互斥量确保系统中      *
;        只有进程的一个拷贝在运行 *
;*********************************
.386p
.model flat,stdcall

;常量定义
NULL    equ 0
MB_OK equ 0
TRUE    equ 1
FALSE equ 0
ERROR_ALREADY_EXISTS          equ 183

;外部过程声明
extrn CreateMutexA:proc
extrn GetLastError:proc
extrn CloseHandle:proc
extrn MessageBoxA:proc
extrn ExitProcess:proc

.data
     Caption db 'Mutex',0
     Text        db 'OK',0
     ExistText db '已有一个本程序的实例在运行！',0
     Mutex      db 'Mutex Object',0
     MutexHandle  dd ?
.code
```

```
main:
    call CreateMutexA,NULL,TRUE,offset Mutex    ;建立互斥对象
    mov  [MutexHandle],eax
    call GetLastError
    cmp  eax,ERROR_ALREADY_EXISTS       ;已存在?
    jz   MutexExist
    call MessageBoxA,NULL,offset Text,offset Caption,MB_OK
    call CloseHandle,[MutexHandle]
    call ExitProcess,0
MutexExist:
    call MessageBoxA,NULL,offset ExistText,offset Caption,MB_OK
    call CloseHandle,[MutexHandle]
    call ExitProcess,0
    end main
```

5.2 WinMain 函数的入口

若你用 C 写过 Win32 应用程序,就应该知道 WinMain()函数(类似于 DOS 下的 main())有四个入口参数。WinMain 的声明如下:

WinMain(HINSTANCE hinst, HINSTANCE prehinst,char *commandLine,int n);

其中的 hinst 和 prehinst 都称为"实例句柄",前者为当前的实例名柄,后者为前一个实例名柄。这实际上是从 Win16 时代遗留下来的概念。在 Win16 中,相同程序的每个实例得到的实例句柄是不同的,可以通过前一实例的句柄 prehinst 是否为 0 来区分是否已有一个本程序的实例在运行。在 Win32 中,程序间若不通过特别的手段是不能知道对方的存在的,参数 prehinst 会恒为 NULL。之所以保留这个参数,仅仅是为了保持源程序的兼容性。而 hinst 则是进程装入内存的基地址,也称模块句柄。

我们看下面的例子:

```
#include <windows.h>
APIENTRY WinMain(HINSTANCE hinst, HINSTANCE prehinst,
                 char *commandLine,int n)
{
    char Array[250];
    wsprintf(Array," \
当前实例: %X\n \
上一实例: %d\n \
命令行:   %s\n \
n:        %d",
        hinst,prehinst,commandLine,n);
    MessageBox(NULL,Array,"看看而已",MB_OK);
```

```
        return 0;
}
```

如图 5-2 所示是在 Visual C++ 6 中编译，在 Windows 2000 中运行的结果，我们看到它的当前实例句柄为 400000H，这实际上是程序的起始地址；上一实例为 0（NULL）。

图 5-2

我们再来看看反汇编的结果：

```
00401000    mov     eax,[esp+10]
00401004    mov     ecx,[esp+0c]
00401008    mov     edx,[esp+08]
0040100c    sub     esp,000000fc
00401012    push    eax                 ;n
00401013    mov     eax,[esp+00000104]
0040101a    push    ecx                 ;命令行：commandline
0040101b    push    edx                 ;上一实例：prehinst
0040101c    push    eax                 ;当前实例：hinst
0040101d    lea     ecx,[esp+10]
00401021    push    0040402c
00401026    push    ecx
00401027    call    [USER32!wsprintfA]
0040102d    add     esp,18
00401030    lea     edx,[esp+00]
00401034    push    00
00401036    push    00404020
0040103b    push    edx
0040103c    push    00
0040103e    call    [USER32!MessageBoxA]
00401044    xor     eax,eax
00401046    add     esp,000000fc
0040104c    ret     0010
```

从上面我们可以找到进入 WinMain()时四个参数所存放的地方。但要注意的是，用 C/C++（或其他任何一种高级语言）编程时，WinMain()并不是程序运行时就马上执行的。在执行 WinMain()之前，程序会执行一段由编译器生成的初始化代码，WinMain()的这些

参数是由这个初始化代码生成的。这个初始化代码调用 GetModuleHandle 来取得模块的实例句柄，并传给 WinMain。

5.3 建立线程和进程

我们可以在自己的进程中创建一个新的进程如图 5-3 所示。建立一个进程的方法不止一种，按旧的 Win16 的方法是使用 Windows API 函数 WinEexc，但时，Win32 进程应该使用 API 函数 CreateProcessA，因为它提供了更多的参数可供选择。

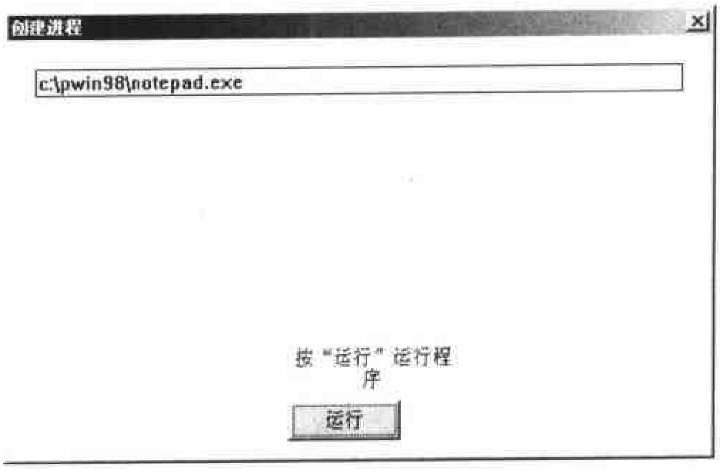


图 5-3

```
;************************
;文件：CreatPro.asm      *
;功能：演示如何创建进程  *
;************************
IDC_PROCESS   = 100
MEM_COMMIT    = 1000H
PAGE_EXECUTE_READWRITE = 40H
.386p
locals
.model flat,stdcall
include win32.inc
include CreatPro.inc
extrn MessageBoxA:proc
extrn ExitProcess:proc
extrn DialogBoxParamA:proc
extrn EndDialog:Proc
extrn GetDlgItem:proc
extrn GetDlgItemTextA:proc
extrn SetDlgItemTextA:proc
```

```
extrn CreateProcessA:proc
extrn CloseHandle:proc
extrn VirtualAllocEx:proc
extrn VirtualFreeEx:proc
extrn GetModuleHandleA:proc
.data
      SUI STARTUPINFO<17*4,>
      PI   PROCESS_INFOMATION<>
      DialogName   db "MyDialog",0
      CaptionFail   db '失败',0
      TextFail  db 'API:DialogBoxParamA 调用失败！ ',0
      TextCPA       db '创建进程失败！ ',0
      FileNamedb 'c:\pwin95\notepad.exe',0
      db 1000 dup(?)
.code
WinMain:
      call GetModuleHandleA,0            ;取模块句柄
      call DialogBoxParamA,eax,offset DialogName,0,offset DialogProc,0
      cmp eax,0                          ;调用 DialogBoxParamA 失败？
      jnz   Exit
      mov eax,offset TextFail            ;出错信息
      call MessageBoxA,0,eax,offset CaptionFail,MB_OK
Exit:call ExitProcess,0
;消息处理函数
DialogProc   proc uses ebx edi esi, hWnd:DWORD, wMsg:DWORD, wParam:DWORD,
lParam:DWORD
      cmp wMsg,WM_INITDIALOO
      jz WmInitDialog
      cmp wMsg,WM_CLOSE
      jz WmClose
      cmp wMsg,WM_COMMAND
      jz WmCommand
         jmp ExitProc
WmInitDialog:
         ;显示程序名
         call SetDlgItemTextA,hWnd,IDC_PROCESS,offset FileName
         jmp ExitProc
WmClose:
         call EndDialog,hWnd,1
```

```
            jmp ExitProc
    WmCommand:
        mov ebx,wParam
        cmp bx,IDOK
        jnz ExitProc
        call GetDlgItemTextA,hWnd,IDC_PROCESS,offset FileName,100    ;取程序名
        call CreateProcessA,offset FileName,0,0,0,FALSE,0,0,0,offset SUI,offset PI
        or eax,eax                  ;创建进程失败
        mov ebx,offset TextCPA
        jz ErrMsg
        call CloseHandle,PI.hProcess
        call CloseHandle,PI.hThread
        jmp ExitProc
        ErrMsg:
        call MessageBoxA,0,ebx,offset CaptionFail,MB_OK
        jmp ExitProc
    ExitProc:
        xor eax,eax
        ret
    DialogProc endp
    end WinMain
```

Win32 的线程和某些（不是全部）UNIX 系统的线程不同，某些 UNIX 系统并不是在操作系统内核中而是通过库函数实现对线程的支持的。但 Win32 却真正地从内核级实现了对多线程的支持，我们可以通过直接调用 Win32 API 来创建、启动、终止线程而无需外加库函数支持。

```
;************************
;文件：Thread.asm        *
;功能：演示线程的创建   *
;      和终止           *
;************************
.386p
locals
.model flat,stdcall
include win32.inc
include api32.inc
.data
    Text        db '好吗',0
    ThreadMainBegin        db '主线程开始',0
```

```
        ThreadMainEnd          db '主线程结束',0
        ThreadBegin1           db '线程 1 开始',0
        ThreadEnd1             db '线程 1 结束',0
        Thread1ID      dd   ?
        hThread1       dd   ?
.code
main:
        call MessageBoxA,0,offset ThreadMainBegin,offset Text,MB_OK
        call CreateThread,0,000H,offset Thread1,0,0,offset Thread1ID
        mov    hThread1,eax
        call MessageBoxA,0,offset ThreadMainEnd,offset Text,MB_OK
        call ExitProcess,0
Thread1:
        call MessageBoxA,0,offset ThreadBegin1,offset Text,MB_OK
        call MessageBoxA,0,offset ThreadEnd1,offset Text,MB_OK
        call TerminateThread,hThread1,0
        end main
```

5.4 线程调度和使用

Win32 的进程由一个主线程开始，而线程又可以创建另一个线程。因此，一个进程可能有多个线程，而系统中又可能有多个进程，再加上系统本身的线程，所以，系统中一般都有多个线程在运行。

Windows NT 的支持工具中有一个程序叫 Process Viewer，如图 5-4 所示。它可以显示出一个进程有几个线程，这些线程在内核态和用户态各花了多少时间等。

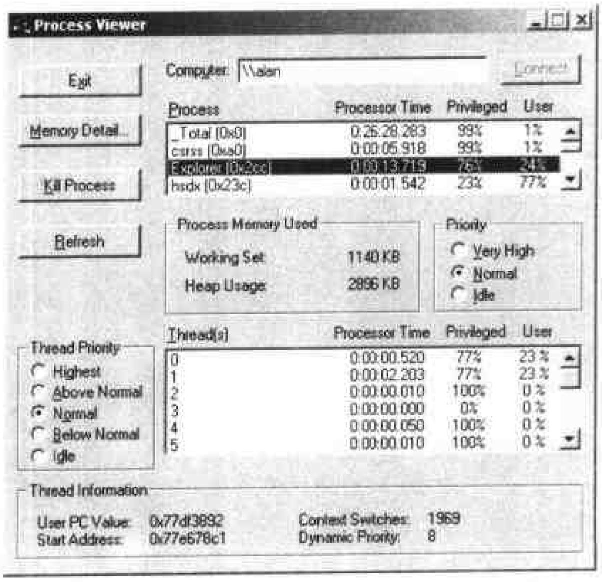


图 5-4

我们知道，线程只是看上去在同时运行，这是操作系统在划分 CPU 时间片分配给各个线程而给我们造成的假象。实际上 CPU 每一个确定的时刻只能运行一个线程，因此，若你的机器有一个 CPU，则某一确定的时刻只能有一个线程在运行，如果有 2 个 CPU，则最多只能有 2 个线程同时运行。

线程可能处于的状态在各系统中各不相同。在 Windows 95 中线程可能处于三种状态之一：运行、就绪、等待，它的意义就如经典操作系统教程中关于进程状态的描述一样，如图 5-5 所示。

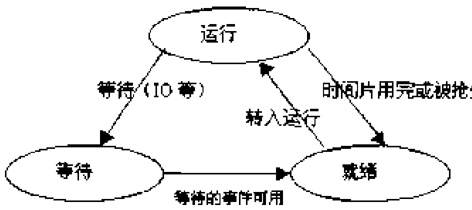


图 5-5

1）就绪：表示线程准备好继续执行。

2）运行：运行中。

3）等待：因 IO 操作、同步的需要而等待资源。

而 WIN NT4 中线程可能处于六种状态之一：就绪、备用、运行、等待、转换、终止。其意义如下：

1）就绪：表示线程准备好继续执行。

2）备用：下一个要执行的线程。

3）运行：运行中。

4）等待：因 IO 操作、同步的需要而等待资源。

5）转换：线程已可以执行但其内核堆栈被换出了内存时。

6）终止：线程已执行完，但可能仍未被系统删除。

其转换图如图 5-6 所示。

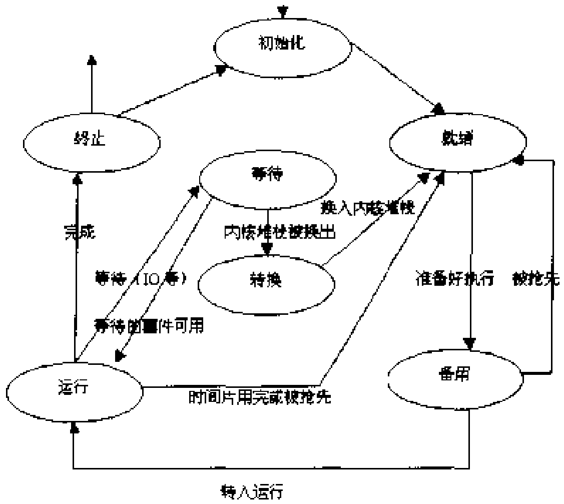


图 5-6

当线程在进程内部转换时，转换所占用的 CPU 时间是比进程间转换少很多的，这是因为线程在进程间转换时，只需要切换通用寄存器和堆栈指针等，而不需换地址空间——即是说，不用重新映射进程的内存地址。

当线程切换时，要保存当前的所有通用寄存器和堆栈指针等东西。要注意，每一个线程都有它自己的堆栈。为什么要这样呢？这是因为每一个线程都要使用堆栈来创建局部变量、传递参数，可以想象，若一个进程的所有线程都使用同一个堆栈，而这些线程随时都可能被抢先和被运行，则堆栈的使用必定一片混乱！

线程在初始化时一般默认保留 1MB 的堆栈，但并不是线程一创建的时候就马上在物理上实际分配 1MB 的内存给它，而是默认只分配 2000HB，即 2 页的内存给堆栈，剩下的部分只有当线程需要时才分配给它。如图 5-7 所示，当线程创建时，提交 2 个页面给线程，一个页面是可读可写的，堆栈指针 ESP 指向了这个页面；另一个页面在比第一个面页低的地址上，它是受保护的。

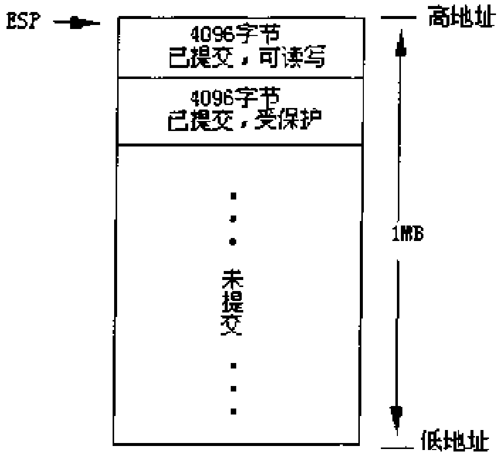


图 5-7

当线程使用堆栈来存放返回地址、生成局部变量时，ESP 指针变小，靠近第二个页面。当最终访问到第二个已提交的页面时，由于第二个页面是受保护的，因此，访问它引发异常，操作系统取得控制权，系统就把这个页面改成可读写的，并且又提交一个虚拟地址上相连的页面，并设为受保护的。当 ESP 又减小到第 3 页处时，又发生与接近第 2 页时一样的情况，系统再提交一页，并设为受保护。依此类推。

Windows 这样处理线程堆栈，系统就不需一次提交 1MB 的内存给线程做堆栈，但又在线程需要时能分配内存给它。

但是，这样也会引起另一个问题：当在一个子程序中用“sub esp,n”指令来创建局部变量时，如果局部变量很大（例如一个 10000 字节的数组），这个变量就可能会跨越可读写的提交页、受保护的提交页和保留未分配的页这三个页面。这时，如果你对这个局部变量存取时先存取低地址的部分，就很可能会先存取未分配的页，这将导致程序因产生异常而终结。实际上，一个变量先存取低地址的部分是很自然的事，例如在 C 语言下面的语句：

```
char array[10000];
array[0] = 'A';
```

其中的 array[0]就是变量的低地址部分。因此，在子程序中使用大的变量时应该十分小心。可以考虑用全局变量而不是局部变量，或保证至少先从高地址向低地址访问一次，然后再随机访问。

在这里，我们本来要提供一个范例程序，但是，由于程序要用到结构化异常处理的知识，因此，将在下面章节提供这个程序。

当线程相互切换时，要保存当前的所有通用寄存器和堆栈指针等线程间独立的信息。不过，我们也可以通过自己调用 API 来自己保存/恢复线程的寄存器内容：

```
FLOATING_SAVE_AREA STRUC
    ControlWord   DD ?
    StatusWord    DD ?
    TagWord       DD ?
    ErrorOffset   DD ?
    ErrorSelector DD ?
    DataOffset    DD ?
    DataSelector  DD ?
    RegisterArea  DB SIZE_OF_80387_REGISTERS DUP(?)
    Cr0NpxState   DD ?
ENDS

CONTEXT STRUC
    ContextFlags DD ?

    X86Dr0  DD ?
    X86Dr1  DD ?
    X86Dr2  DD ?
    X86Dr3  DD ?
    X86Dr6  DD ?
    X86Dr7  DD ?

    FloatSave FLOATING_SAVE_AREA<>

    SegGs   dd ?
    SegFs   dd ?
    SegEs   dd ?
    SegDs   dd ?

    X86Edi  dd ?
    X86Esi  dd ?
    X86Ebx  dd ?
```

```
        X86Edx  dd ?
        X86Ecx  dd ?
        X86Eax  dd ?

        X86Ebp  dd ?
        X86Eip  dd ?
        SegCs   dd ?
        EFlags  dd ?
        X86Esp  dd ?
        SegSs   dd ?
    ENDS

        hThread   dd  ?
        lpContext   CONTEXT<>
        ......
    call   SuspendThread,hThread
        call GetThreadContext,[hThread],offset lpContex          ;保存
        cmp eax,0
        jz Error
        ......
    call   SuspendThread, hThread
        call SetThreadContext,[hThread],offset lpContex          ;设置
        cmp eax,0
        jz Error
        ......
    Error:
        ......
```

这两个 API 保存和设置指定线程的寄存器内容，包括通用寄存器、系统寄存器、浮点部件 80387 的寄存器等。

注意若你想保存当前线程的状态，这个函数是不够的，因为它只是保存了寄存器的内容，并没有保存局部变量的值。另外，你应该先挂起这个线程序再保存它的寄存器内容。

下面是一个使用线程的例子。我们知道，MessageBox 调用会阻塞线程，直到用户按下“确定”按钮，使我们的程序不容易对它进行控制。我们这里用新建一个线程的方法，在调用 MessageBoxA 之前建立一个线程，这个线程的工作是用 FindWindowA 找出 MessageBox 的句柄，然后用 SetWindowLong 来改变这个 MessageBox 的消息处理函数，使得用户按下 MessageBox 上的“确定”按钮时，MessageBox 并不马上退出，而是要用户按 10 次按钮时才退出。

FindWindowA 的调用如下：

HWnd dd ?

call FindWindowA,offset ClassName,offset WindowName

mov [hWnd],eax

其中：

1）CallName 通常为类名，也可以是标识类的原子。

2）WindowName 是窗口的名称（标题）。

SetWindowLongA 和 GetWindowLongA 取得或设置窗口的指定属性：

Long dd ?

call GetWindowLongA,hWnd,index

mov [Long],eax

call SetWindowLongA,hWnd,index,NewLong

其中：

1）hWnd 是要取得或改变的窗口的句柄。

2）Index 是要改变或取得的内容，可以是以下的值之一：

GWL_EXSTYLE，窗口扩展风格。

GWL_STYLE，窗口风格。

GWL_WNDPROC，窗口处理函数，这正是我们这个程序需要的。

GWL_HINSTANCE，窗口所属的模块实例句柄。

GWL_HWNDPARENT，父窗口句柄。

GWL_ID，窗口 ID。

GWL_USERDATA，与窗口相联的一个 32 位值。

3）NewLong 是新的值。

如图 5-8 所示是程序运行的结果。

（a）程序刚启动时　　（b）点击过 5 次后

图 5-8

线程启动时先取得 MessageBox 原来的消息处理函数地址，然后设置它的消息处理函数成为我们写的函数。

在我们的消息处理函数中，判断是否点击了"确定"按钮，如果"是"，则改变 MessageBox 的文本内容，否则调用原来旧的处理函数。

下面是程序的源代码，还需说明的是程序中定义了两个常量：

MB_OK_ID = 2

MB_STATIC_ID = 0FFFFH

它们分别是 MessageBox 的"确定"按钮和静态文本的 ID。这两个值在 SetDlgItemTextA 和判断是消息是否点击"确定"按钮时有用。

```
;**************************
;文件：SetFun.asm           *
;功能：用线程异步地改变      *
;        窗口的处理函数      *
;**************************

.386p
.model flat,stdcall
include win32.inc
extrn MessageBoxA:proc
extrn CreateThread:proc
extrn TerminateThread:proc
extrn ExitProcess:proc
extrn FindWindowA:proc
extrn Sleep:proc
extrn SetWindowLongA:proc
extrn GetWindowLongA:proc
extrn ExitThread:proc
extrn SetDlgItemTextA:proc

NULL   = 0
MB_OK = 0
GWL_WNDPROC     =   -4
MB_OK_ID   = 2
MB_STATIC_ID   = 0FFFFH

.data
    Welcome db '你按了“确定”或“X”按钮'
    Times    db 30H
             db '次',0
    Text     db '您好！请按确定 10 次退出程序！',0
    OK       db 'This is a sample',0
    OldProc  dd ?

    ThreadID dd ?
    hThread  dd ?

.code
main:
```

```
        call CreateThread,NULL,0,offset ThreadProc,0,0,offset ThreadID
        mov   hThread,eax

        call MessageBoxA,0,offset Text,offset OK,MB_OK
        call ExitProcess,0

    ;线程处理函数
    ThreadProc proc
    ThreadBegin:
        call FindWindowA,NULL,offset OK
        or    eax,eax                   ;窗口存在否
        jnz   change
        call Sleep,500
        jmp   ThreadBegin
    change:
        push eax
        call GetWindowLongA,eax,GWL_WNDPROC    ;设置窗口过程
        mov   [OldProc],eax
        pop   eax
        call SetWindowLongA,eax,GWL_WNDPROC,offset NewProc
    Error:
        ret
    ThreadProc endp

    ;新的 MessageBox 窗口处理函数
    NewProc proc uses ebx edi esi, handle:DWORD, wMsg:DWORD, wParam1:DWORD,
lParam1:DWORD
        cmp   wMsg,WM_COMMAND
        jz    command
        jmp   old
    command:
        mov   ebx,wParam1
        cmp   bx,MB_OK_ID
        jnz   old
        inc   [Times]                   ;点击次数加 1
        cmp   [Times],3AH               ;够 10 次?
        jz    old
        call SetDlgItemTextA,handle,MB_STATIC_ID,offset Welcome
        ret
```

```
old:
    mov   eax,[OldProc]
    call eax,handle,wMsg,wParam1,lParam1
    ret
NewProc endp
    end main
```

下面我们来看一个稍复杂又更实用的例子。这个例子利用了线程的网络编程例子。

这 2 个 Socket 程序是典型的 Server/Client 结构模式。在 Server 中，开始的时候另外建立一个线程来监听客户端的连接。它是一个死循环，在每次接受连接时，又开一个新的线程来处理这次连接的任务，然后又继续接受下一次连接。接受连接时新开出的线程向 client 发出一个字符串，告知这是第几次连接。

服务器的主线程弹出一个 MessageBox，当用户按下“确定”时，程序结束。

客户端的动作是连上服务器，读取一个字符串，然后将其显示出来。

程序运行结果如图 5-9 所示。

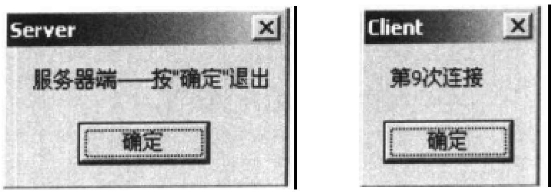


图 5-9

```
;************************************
;文件：Server.asm                    *
;功能：演示线程的使用的 Socket 编程    *
;          ——服务器端               *
;************************************
.386p
.model flat,stdcall
include win32.inc
extrn MessageBoxA:proc
extrn ExitProcess:proc
extrn WSAStartup:proc
extrn htons:proc
extrn gethostbyname:proc
extrn getprotobyname:proc
extrn socket:proc
extrn connect:proc
extrn recv:proc
extrn closesocket:proc
extrn accept:proc
```

```
extrn _wsprintfA:proc
extrn send:proc
extrn bind:proc
extrn listen:proc
extrn CreateThread:proc
AF_INET         =  2
PF_INET                 =  AF_INET
SOCK_STREAM             =  1
SOCK_DGRAM              =  2
SOCK_RAW                =  3
SOCK_RDM                =  4
SOCK_SEQPACKET=  5
INADDR_ANY                 =     00000000H
INADDR_LOOPBACK            =     7f000001H
INADDR_BROADCAST           =   0ffffffffH
INADDR_NONE                =    0ffffffffH
ADDR_ANY                   =     INADDR_ANY

NULL   = 0
MB_OK = 0
PORT = 1357
QLEN = 6

in_addr   struc
     s_addr     dd ?
     ends

hostent   struc
          h_name          dd ?
          h_aliases dd ?
          h_addrtype      dw ?
          h_length  dw ?
          h_addr_list     dd ?
;h_addr   h_addr_list[0]             /* address, for backward compat */
     ends

protoent  struc
          p_name          dd ?
          p_aliases dd ?
```

```
        p_proto         dw ?
                        dw ?
    ends

sockaddr_in    struc
        sin_family      dw ?
        sin_port   dw ?
        sin_addr   dd ?
        sin_zero   db 8 dup (?)
    ends
WSADESCRIPTION_LEN          = 256
WSASYS_STATUS_LEN           = 128

WSADATA    struc
        wVersion        dw ?
        wHighVersion dw ?
        szDescription  db WSADESCRIPTION_LEN+1 dup (0)
        szSystemStatus     db WSASYS_STATUS_LEN+1     dup (0)
        iMaxSockets         dw ?
        iMaxUdpDg           dw ?
        lpVendorInfo   dd ?
    ends

.data
    Caption   db 'Server',0
    Text      db '服务器端——按"确定"退出',0
    proto     db 'tcp',0
    host      db 'localhost',0      ;20 dup (0)
    strtmp    db '第%d 次连接',0
    align 4
    ptrp      dd ?
    alen      dd ?
    sd        dd ?
    port      dd ?
    visitors  dd ?
    ThreadID dd ?
    align 4
    sad       sockaddr_in<>
    align 4
```

```
        cad         sockaddr_in<>
        align 4
        wsaData  WSADATA<>

.code
main:
        call  CreateThread,NULL,0,offset ThreadProc,0,0,offset ThreadID

        ;主线程按"确定"会退出
        call        MessageBoxA,0,offset Text,offset Caption,MB_OK
        call        closesocket,sd
        call        ExitProcess,0

ThreadProc:
        call  WSAStartup,0101H,offset wsaData      ;为使用 WS2_32.DLL 初始化

        mov         sad.sin_family,AF_INET
        mov         sad.sin_addr,INADDR_ANY
        mov         port,PORT
        mov         eax,port
        push        eax
        call        htons                          ;主机字节顺到网络字节顺序转换
        mov         sad.sin_port,ax
;映射协议名到协议号
        call        getprotobyname, offset proto
        mov         ptrp,eax

        movzx       eax,[eax].p_proto
        call        socket,PF_INET,SOCK_STREAM,eax
        mov         sd,eax
        call        bind,sd,offset sad,size sad    ;绑定

        call        listen,sd,QLEN                 ;监听

acceptnext:
        mov         alen,size cad
        ;接受连接
        call        accept,sd,offset cad,offset alen
        ;另开线程处理连接
```

```
        call    CreateThread,NULL,0,offset Service,eax,0,offset ThreadID
        jmp     acceptnext

Service proc sd2:DWORD
        inc     visitors                ;访问数加 1
        sub     esp,100

        mov     ebx,esp                 ;格式化字符串
        call    _wsprintfA,ebx,offset strtmp,visitors
        add     esp,3*4
        call    send,sd2,ebx,eax,0      ;发送
        call    closesocket,sd2         ;关闭
        add     esp,100
        ret
Service endp
end main
```

下面是客户端程序，它连上，读取服务器传来的字符串，并用 Messagebox 将其显示出来。

```
;************************************
;文件：Client.asm                    *
;功能：演示线程的使用的 Socket 编程    *
;         ——客户端                   *
;************************************
.386p
.model flat,stdcall
extrn MessageBoxA:proc
extrn ExitProcess:proc
extrn WSAStartup:proc
extrn htons:proc
extrn gethostbyname:proc
extrn getprotobyname:proc
extrn socket:proc
extrn connect:proc
extrn recv:proc
extrn closesocket:proc

AF_INET             =  2
PF_INET             =  AF_INET
SOCK_STREAM         =  1
```

```
SOCK_DGRAM      = 2
SOCK_RAW        = 3
SOCK_RDM        = 4
SOCK_SEQPACKET  = 5

NULL   = 0
MB_OK = 0
PORT   = 1357

hostent  struc
          h_name        dd ?
          h_aliases     dd ?
          h_addrtype    dw ?
          h_length      dw ?
          h_addr_list   dd ?
;h_addr  h_addr_list[0]              /* address, for backward compat */
     ends

protoent  struc
          p_name        dd ?
          p_aliases     dd ?
          p_proto       dw ?
                        dw ?
     ends

sockaddr_in   struc
          sin_family    dw ?
          sin_port      dw ?
          sin_addr      dd ?
          sin_zero      db 8 dup (?)
     ends

WSADESCRIPTION_LEN            = 256
WSASYS_STATUS_LEN             = 128

WSADATA    struc
          wVersion         dw ?
          wHighVersion     dw ?
          szDescription    db WSADESCRIPTION_LEN+1 dup (0)
```

```
        szSystemStatus      db WSASYS_STATUS_LEN+1    dup (0)
        iMaxSockets         dw ?
        iMaxUdpDg           dw ?
        lpVendorInfo        dd ?
    ends

.data
    Caption         db 'Client',0

    proto           db 'tcp',0
    host            db 'localhost',0
    buf             db 1000 dup (0)

    align 4
    ptrh            dd ? ;point to hostent<>
    ptrp            dd ? ; point to protoent<>
    sd              dd ?
    port            dd ?
    align 4
    sad         sockaddr_in<>
    align 4
    wsaData  WSADATA<>

.code
main:
    ;为使用 WS2_32.DLL 初始化
    call        WSAStartup,0101H,offset wsaData

    mov         sad.sin_family,AF_INET
    mov         port,PORT
    mov         eax,port
    push eax
    call        htons
    mov         sad.sin_port,ax

    call        gethostbyname,offset host     ;取服务器
    mov         ptrh,eax
    mov         ebx,offset sad
    add         ebx,sin_addr
```

```
        movsx   ecx,[eax].h_length
        add     eax,h_addr_list
        mov     eax,[eax]
        mov     eax,[eax]
        call    CopyMemory,ebx,eax,ecx

        call    getprotobyname, offset proto  ;取协议
        mov     ptrp,eax

        movzx   eax,[eax].p_proto
        call    socket,PF_INET,SOCK_STREAM,eax
        mov     sd,eax

        call    connect,eax,offset sad,size sad

        mov     esi,offset buf
recvnext:
        push esi
        call    recv,sd,offset esi,1000,0        ;接收
        pop     esi
        add     esi,eax
        or      eax,eax
        jnz     recvnext

        call    MessageBoxA,NULL,offset buf,offset Caption,MB_OK
MainEnd:
        call    closesocket,sd
        call    ExitProcess,0

CopyMemory proc des:DWORD,src:DWORD,len:DWORD
        mov     edi,des
        mov     esi,src
        mov     ecx,len
        cld
        rep     movsb
        ret
CopyMemory endp
        end main
```

说到 Socket 编程，我们可以顺带说一说 Raw Socket，因为它对我们对机器的特征和网络编程都有帮助。

Raw socket 提供了一种我们直接控制网络数据包的方法。

用一般 socket 函数通讯时，并不需我们自己处理数据包，它使我们不必理会协议如何工作，只管把数据交给 socket 函数就行了。

但有时我们可能会想直接控制数据包的各部分，例如 Ping 程序，装有 TCP/IP 协议栈的机器一般都带有这个程序。

Ping 程序要直接对 ICMP 包进行控制，这就要用 Raw Sock。

建立 Raw Socket 用到下面的语句：

```
call WSASocketA,AF_INET,SOCK_RAW,IPPROTO_ICMP,NULL, 0,0
mov rockRaw,eax
```

它产生一个 Raw Socket，所用协议是 TCP/IP 的 ICMP 协议。

在这个程序中，我们自己直接生成 ICMP 包，并接收原始的 IP 包进行分析，输出分析的内容。

程序中用到 IP 和 ICMP 数据包，它们定义如下：

IP 包头：

```
IpHeader struc
     len_ver    db ?
     tos        db ?
     total_len  dw ?
     ident      dw ?
     frag_and_flags      dw ?
     ttl        db ?
     proto      db ?
     checksum       dw ?
     sourceIP  dd ?
     destIP     dd ?
          ends
```

ICMP 包头：

```
IcmpHeader   struc
  i_type       db ?
  i_code       db ?
  i_cksum      dw ?
  i_id         dw ?
  i_seq        dw ?
          ends
```

这里要注意如下两点：

1）网络字节顺序的问题。在 IP 包头中，第一个字节有版本和头部长的信息，一般的网络书籍在画 IP 头部时版本是放在第 0~3 位，头部长是第 4~7 位。但在我们下面的程序

中认为头部放在 0~3 位，而版本放在 4~7 位。这是因为 X86 机器的字节顺序与网络字节顺序相反，X86 是低位在前，高位在后，称为 little-endian，而网络字节顺序则相反，称为 big-endian。我们写一般 socket 程序指定端口时，要用 htonl 函数来转换也是这个原因。

2）上面的 ICMP 包头有 5 个域，但是，我们下面的程序会给它增加一个域 timestamp，用以记录发送时的时间，在接收时再与当前时间比较，就知道数据包来回用了多少时间了。用 GetTickCount 函数取得时间，这是系统启动到当前的微秒数。但是，由于 Windows 时钟本身分辨率的问题，因此得到的时间差并不准确。

如图 5-10 所示的是命令 ping localhost 的输出。

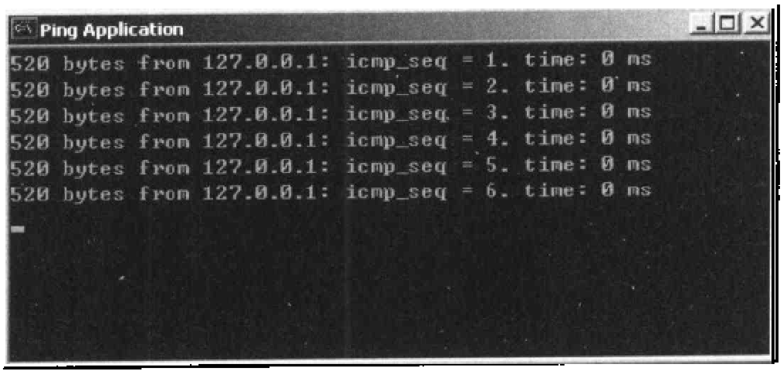


图 5-10

```
;*****************************
;文件：Ping.asm              *
;功能：Ping 程序和 RAW Socket *
;*****************************
.386p
.model flat,stdcall
extrn MessageBoxA:proc
extrn ExitProcess:proc
extrn WSAStartup:proc
extrn htons:proc
extrn gethostbyname:proc
extrn getprotobyname:proc
extrn socket:proc
extrn connect:proc
extrn recv:proc
extrn recvfrom:proc
extrn closesocket:proc
extrn accept:proc
extrn _wsprintfA:proc
extrn send:proc
extrn sendto:proc
```

```
extrn bind:proc
extrn listen:proc
extrn GetTickCount:proc
extrn Sleep:proc
extrn WSASocketA:proc
extrn setsockopt:proc
extrn inet_ntoa:proc
extrn WriteConsoleA:proc
extrn SetConsoleTitleA:proc
extrn CreateConsoleScreenBuffer:proc
extrn CloseHandle:proc
extrn SetConsoleActiveScreenBuffer:proc
extrn GetCommandLineA:proc

NULL    =0
MB_OK = 0
GENERIC_READ        equ  80000000H
GENERIC_WRITE       equ  40000000H
FILE_SHARE_READ     equ  00000001H
FILE_SHARE_WRITE    equ  00000002H
CONSOLE_TEXTMODE_BUFFER equ     1
INVALID_HANDLE_VALUE        equ    -1
IPPROTO_IP      =       0
IPPROTO_ICMP =      1
AF_INET         =   2
PF_INET         =  AF_INET
SOCK_RAW          =  3
SOL_SOCKET     =     0ffffH
SO_SNDTIMEO         =1005H
SO_RCVTIMEO         =1006H
ICMP_ECHO    = 8
ICMP_ECHOREPLY = 0

IpHeader struc
    len_ver         db ?
    tos             db ?
    total_len       dw ?
    ident           dw ?
    frag_and_flags  dw ?
```

```
        ttl             db ?
        proto           db ?
        checksum        dw ?
        sourceIP        dd ?
        destIP          dd ?
                ends

IcmpHeader  struc
    i_type      db ?
    i_code      db ?
    i_cksum     dw ?
    i_id        dw ?
    i_seq       dw ?
    timestamp   dd ?
            ends
in_addr   struc
      s_addr    dd ?
      ends

hostent   struc
            h_name          dd ?
            h_aliases       dd ?
            h_addrtype      dw ?
            h_length        dw ?
            h_addr_list     dd ?
;h_addr   h_addr_list[0]                /* address, for backward compat */
      ends

protoent  struc
            p_name          dd ?
            p_aliases       dd ?
            p_proto         dw ?
                            dw ?
      ends

sockaddr_in    struc
            sin_family      dw ?
            sin_port        dw ?
            sin_addr        dd ?
```

```
        sin_zero        db 8 dup (?)
    ends

WSADESCRIPTION_LEN          = 256
WSASYS_STATUS_LEN           = 128
WSADATA    struc
        wVersion            dw ?
        wHighVersion        dw ?
        szDescription       db WSADESCRIPTION_LEN+1 dup (0)
        szSystemStatus      db WSASYS_STATUS_LEN+1    dup (0)
        iMaxSockets   dw ?
        iMaxUdpDg     dw ?
        lpVendorInfo  dd ?
    ends

.data
    Caption db 'Ping Application',0

    Text db 100 dup (?)
    tmp         db '%d bytes from %s: icmp_seq = %d. time: %d ms',13,10,0
    align 4
    hCS         dd ?
    host        dd ?
    ph          dd ?
    sockRaw dd ?
    timeout     dd ?
    fromlen     dd ?
    wsaData     WSADATA<>
    from        sockaddr_in<>
    desc        sockaddr_in<>

ICMP_DATA_LEN = 500
ICMP_DATA_RECV_LEN = ICMP_DATA_LEN*2
    icmp_data       db ICMP_DATA_LEN    dup (0)
    recvbuf         db ICMP_DATA_RECV_LEN dup (0)

.code
main:
    call        SetConsoleTitleA ,offset Caption    ;改变窗口标题
```

```
        call  CreateConsoleScreenBuffer, GENERIC_WRITE, FILE_SHARE_WRITE, NULL,
CONSOLE_TEXTMODE_BUFFER, NULL
        cmp     eax,INVALID_HANDLE_VALUE          ;建立失败?
        jz      Error
        mov     [hCS],eax
        call    SetConsoleActiveScreenBuffer,eax    ;设置有效的制台 screen buffer
    ;*****************************************************************
        ;读命令行
        call    GetCommandLineA
        mov     esi,eax
        cld
    load:
        lodsb
        cmp     al,' '
        jz      GetCommandLineNext
        cmp     al,0
        jz      GetCommandLineNext
        jmp     load
    GetCommandLineNext:
        cmp     al,0
        jz      Error
    load1:
        lodsb
        cmp     al,' '
        jz      load1
        dec     esi
        mov     host,esi

        call    WSAStartup,0201H,offset wsaData
        ;建立 Raw Socket
        call    WSASocketA,AF_INET,SOCK_RAW,IPPROTO_ICMP,NULL, 0,0
        mov     sockRaw,eax

        ;设置超时
        mov     timeout,1000
        call    setsockopt, sockRaw, SOL_SOCKET, SO_RCVTIMEO, offset timeout, size
timeout
        mov     timeout,1000
        call    setsockopt,sockRaw,SOL_SOCKET,SO_SNDTIMEO,offset    timeout,size
```

```
timeout
        call    gethostbyname, host
        mov     ph,eax
        or      eax,eax
        jz      Error
        movzx   ecx,[eax].h_length
        mov     edx,[eax].h_addr_list
        mov     edx,[edx]
        mov     ebx,offset desc
        add     ebx,sin_addr
        call    CopyMemory,ebx,edx,ecx

        mov     ebx,ph
        mov     ax,[ebx].h_addrtype
        mov     desc.sin_family,ax

        ;填 icmp 包数据
        mov     icmp_data.i_type,ICMP_ECHO
        mov     icmp_data.i_code,0
        mov     icmp_data.i_id,1234H
        mov     icmp_data.i_seq,0
        mov     ebx,offset icmp_data
        add     ebx,size IcmpHeader
        call    SetMemory,ebx,'Q',ICMP_DATA_LEN - size IcmpHeader

pingagain:
        mov     icmp_data.i_cksum,0
        call    GetTickCount
        mov     icmp_data.timestamp,eax
        inc     icmp_data.i_seq
        call    CalcSum,offset icmp_data,ICMP_DATA_LEN
        mov     icmp_data.i_cksum,ax
        ;送出包
        call    sendto,sockRaw,offset icmp_data,ICMP_DATA_LEN,0,offset desc,size desc
        cmp     eax,0
        js      Error
        mov     fromlen,size from
        ;接收
        call    recvfrom, sockRaw, offset recvbuf, ICMP_DATA_RECV_LEN, 0, offset
```

```
from, offset fromlen
        cmp     eax,0
        js      Error
        ;分析并打印包信息
        call    decode_resp,eax

        call    Sleep,1000
        jmp     pingagain
    Error:
    MainEnd:
        call    CloseHandle,[hCS]
        call    ExitProcess,0
    ;************************************
    CopyMemory proc des:DWORD,src:DWORD,len:DWORD
        mov     edi,des
        mov     esi,src
        mov     ecx,len
        cld
        rep     movsb
        ret
    CopyMemory endp

    SetMemory proc p:DWORD,b:DWORD,n:DWORD
        mov     edi,p
        mov     eax,b
        mov     ecx,n
        cld
        rep     stosb
        ret
    SetMemory endp
    ;*************************************
    ;计算校验和
    CalcSum proc buff:DWORD,len:DWORD
        mov     ecx,len
        mov     ebx,buff
        xor     eax,eax
        xor     esi,esi
    calc:
        mov     si,[ebx]
```

```
        add     eax,esi
        add     ebx,2
        sub     ecx,2
        jnz     calc

        mov     ebx,eax
        shr     ebx,16
        movzx   eax,ax
        add     eax,ebx

        mov     ebx,eax
        shr     ebx,16
        add     eax,ebx

        not     ax
        ret
CalcSum endp
;分析并打印收到的包信息
decode_resp proc bytes:DWORD
        LOCAL databegin:DWORD
        mov     bl,recvbuf.len_ver
        and     bl,0fh
        movzx   ebx,bl
        shl     ebx,2
        add     ebx,offset recvbuf
        mov     databegin,ebx

        mov     ecx,offset from
        add     ecx,sin_addr
        mov     ecx,[ecx]
        call    inet_ntoa,ecx
        push eax
        call    GetTickCount
        mov     ebx,databegin
        sub     eax,[ebx].timestamp
        pop     edx
        movzx   ecx,[ebx].i_seq
        call    _wsprintfA,offset Text,offset tmp,bytes,edx,ecx,eax
        add     esp,4*6
```

```
        call    print,offset Text,eax
        ret
decode_resp endp
;*********************************************
print proc what:DWORD,len:DWORD
        LOCAL   Written:DWORD
        lea     ebx,Written
        call    WriteConsoleA,[hCS],what,len,ebx,NULL
        ret
print endp
end main
```

5.5 写入远程进程

在 32 位 Windows 中，各进程有自己私有的空间，各进程间互不干扰。也就是说，当两个不同的进程有一个相同值的指针，当它们读指针所指向的内容时，也会得到不同的内容，这一点有利于现在 CPU 的分页机制。一般情况下，一个进程不能，也不应该读写另一个进程的内存地址空间。但是，在 Windows NT 中，可以通过一些技巧来进入其他进程的地址空间，并且可在它里面建立一个线程来执行我们的指令流。

要写入远程进程空间，先要在远程进程中分配内存，这可以用 Windows NT 提供的 VirtualAllocEx 函数，Windows 95/98 不支持这个函数。

VirtualAllocEx 函数的用法如下：

```
Mem dd ?
call VirtualAllocEx, hProcess, Addr,Size, AllocType,Protection
or eax,eax
jz error
mov [Mem],eax
```

其中：

1）hProcess 是要分配内存的进程句柄。

2）Addr 是要求分配的地址，可设为 NULL，让函数自己决定。

3）Size 是要分配的内存的尺寸。

4）AllocType 指定如何分配内存，与 VirtualAlloc 中的相应参数基本相同。

5）Protection 表明内存的保护方式，与 VirtualAlloc 中的相应参数基本相同。

函数成功则返回分配的内存地址，否则返回 NULL。

内存分配成功后，我们就要把数据和代码写入远程进程中。VirtualAllocEx 返回了内存的指针，但不能直接往这个指针中写内容，因为这个指针指向的是远程进程地址空间中的地址，我们必须用别的方法来写这个地址。在 Win NT 中可以通过 WriteProcessMemory 来写远程的进程的地址空间。例如：

```
call WriteProcessMemory, .hProcess, Addr, offset Buffer, Size,offset cWritten
```

其中：

1）hProcess 是要写入的进程的句柄。

2）Addr 是远程进程中的地址，即要写入的地址。

3）Buffer 是本地地址，即要写的数据。

4）Size 是要求写的大小。

5）cWriten 是函数用以填写实际写了的字节数。

把数据和代码写入远程进程后，就要执行它们了。这需要在远程进程中建立一个线程，在 Windows NT 中，可以用函数 CreateRemoteThread 来实现：

call CreateRemoteThread, hProcess, offset Security, StackSize, offset Func ,Param , offset CreateFlag, offset Threadid

其中：

1）hProcess 是远程进程的句柄。

2）Security 是一个描述线程安全的结构，可置为 NULL。

3）StakSize 是线程堆栈的大小。

4）Func 是线程的处理函数。

5）Param 是线程的参数。

6)CreateFlag 是建立的标志，若为 CREATE_SUSPENDED 则线程建立后挂起，为 NULL 则立即运行。

7）Threadid 是函数用以保存线程 ID 的地址。

现在似乎一切完备了，但事情并非如此简单，首先一个问题是内存寻址的问题。若访问的是堆栈中的局部变量，那很好办，因为那是相对于指针 esp 或 ebp 的相对寻址，但是若要访问一个全局变量，那又怎样？因为分配的地址是可变的，因此线程处理函数中必须动态定位自己的位置，再通过这个位置的偏移来取得变量的值。我们用以下指令序列来取得变量的地址：

```
call Begin          ; 跳到 Begin 处，并使 var 的地址入栈
var  dd ?
……
Begin:
    pop ebx         ; 取得 var 的地址
    mov eax,[ebx]   ; 取得 var 中的值
```

当然也可以在调用线程时把基地址传给线程处理函数，但是没有这种方法直接。

另一问题是：若在远程线程处理函数中调用了 API 函数，也会出问题。因为在调用 API 时，编译器并不生成直接调用 API 的指令，而是在进程装入时根据程序的要求，在一个地址中写入对应 API 的地址，call 指令再根据这个地址调用真正的 API 函数。但是，每个进程中放有的相应 API 的地址并不相同，而且也未必有相应 API 入口地址（因为远程进程未必需要这个 API），因此，我们必须自己找出 API 的真实地址，然后把这个地址也送到远程地址空间中去。这个并不难办，只要用 GetModuleHandleA 和 GetProcAddress 可取得，例如：

```
FuncAddr dd ?
```

```
call GetModuleHandleA,offset ModuleName
call GetProcAddress,eax, offset FunctionName
mov [FuncAddr],eax
```

它们的参数比较容易理解，这里不再细说。

下面是一个示例程序，它产生如图 5-11 所示的结果。

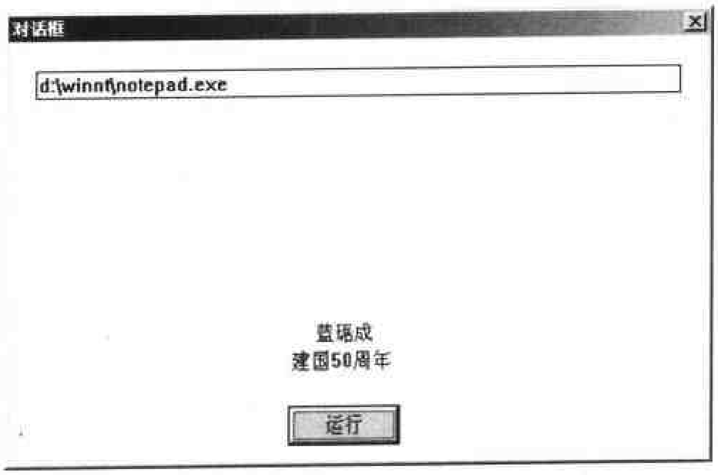


图 5-11

当单击“运行”时，则建立相应的进程，并把我们的代码注入这个进程中，然后运行它，则产生如图 5-12 的结果，我们的线程产生一个 MessageBox 框。

图 5-12

由于我们产生的线程不是注入进程的主线程，因此，当你先结束远程线程（在这里是 Notepad.exe），MessageBox 框也会跟着消失，反之则不会。

```
;********************************
;文件：WritePr.asm              *
;功能：进入另一个进程的地址空间  *
;********************************
.386p
locals
.model flat,stdcall
include win32.inc
include WritePr.inc
extrn MessageBoxA:proc
extrn ExitProcess:proc
extrn DialogBoxParamA:proc
```

```
extrn EndDialog:Proc
extrn GetDlgItemTextA:proc
extrn SetDlgItemTextA:proc
extrn CreateProcessA:proc
extrn CloseHandle:proc
extrn VirtualAllocEx:proc
extrn VirtualFreeEx:proc
extrn GetModuleHandleA:proc
extrn GetProcAddress:proc
extrn WriteProcessMemory:proc
extrn CreateRemoteThread:proc

IDC_PROCESS      = 100
MEM_COMMIT       = 1000H
PAGE_EXECUTE_READWRITE = 40H

.data
     SUI  STARTUPINFO<17*4,>
     PI   PROCESS_INFOMATION<>
     DialogName   db "MyDialog",0
     CaptionFail  db '失败',0
     TextFail     db 'API:DialogBoxParamA 调用失败！ ',0
     TextCPA      db '创建进程失败！ ',0
     TextVAE      db '分配远程内存失败！ ',0
     TextWPM      db '写远程进程失败!',0
     TextCRT      db '建立远程线程失败！ ',0
          FileName db 'd:\winnt\notepad.exe',0
                   db 300 dup(?)

     hKer32       dd ?
     szKer32      db 'KERNEL32.DLL',0
     hUsr32       dd ?
     szUsr32      db 'USER32.DLL',0
     szMsgBox     db 'MessageBoxA',0
     szExitThread db 'ExitThread',0
     InitCaption  db '初始化',0
     InitText     db '初始化失败',0

     MemBase          dd ?
```

```
        NumWritten      dd ?
        hThreadNew      dd ?
        dwThreadNew     dd ?
    RemoteThreadProc:
            call  @0
    MsgBoxAddr       dd ?
    ExitThreadAddr   dd ?
    ThreadCaption db 'Wellcome',0
    ThreadText       db '现正在远程进程地址空间运行',0
    @0: pop     ebx
        mov     eax,[ebx]
        mov     esi,[ebx+4]
        push    esi
        lea     esi,[ebx+4*2]
        lea     edi,[ebx+ThreadText-MsgBoxAddr]
        call  eax,0,edi,esi,MB_OK
        pop     esi
        call  esi,0
    RemoteThreadProcLeng=$-RemoteThreadProc
    .code
    WinMain:
        call Init        ;初始化
        call GetModuleHandleA,0
        call DialogBoxParamA,eax,offset DialogName,0,offset DialogProc,0
        cmp   eax,0
        jnz   Exit
        mov   eax,offset TextFail
        call MessageBoxA,0,eax,offset CaptionFail,MB_OK
    Exit:
        call ExitProcess,0
    ;************************************************************
    ;窗口处理
    DialogProc   proc uses ebx edi esi, hWnd:DWORD, wMsg:DWORD, wParam:DWORD,
lParam:DWORD
        cmp   wMsg,WM_INITDIALOG
        jz    WmInitDialog
        cmp   wMsg,WM_CLOSE
        jz    WmClose
        cmp   wMsg,WM_COMMAND
```

```
        jz      WmCommand
        jmp     ExitProc
WmInitDialog:
        call SetDlgItemTextA,hWnd,IDC_PROCESS,offset FileName
        jmp     ExitProc
WmClose:
        call EndDialog,hWnd,1
        jmp     ExitProc
WmCommand:
        mov     ebx,wParam
        cmp     bx,IDOK
        jnz     ExitProc
        call GetDlgItemTextA,hWnd,IDC_PROCESS,offset FileName,200
        call Ingress
        jmp     ExitProc
ExitProc:
        xor     eax,eax
        ret
DialogProc endp
Init Proc
        call GetModuleHandleA,offset szKer32        取 Kernel32.dll 模块句柄
        mov     [hKer32],eax
        or      eax,eax
        jz      Error
        call GetModuleHandleA,offset szUsr32 ;      User32.dll 模块句柄
        mov     [hUsr32],eax
        or      eax,eax
        jz      Error
        call GetProcAddress,[hKer32],offset szExitThread
        mov     [ExitThreadAddr],eax
        or      eax,eax
        jz      Error
        call GetProcAddress,[hUsr32],offset szMsgBox
        mov     [MsgBoxAddr],eax
        or      eax,eax
        jz      Error
        ret
Error:
        call MessageBoxA,0,offset InitCaption,offset InitText,MB_OK
```

```
        call ExitProcess,0
Init endp
Ingress proc
        call CreateProcessA,offset FileName,0,0,0,FALSE,0,0,0,offset SUI,offset PI
        or     eax,eax
        mov    ebx,offset TextCPA
        jz     ErrMsg
        ;在远程地址空间分配内存
        call VirtualAllocEx, PI.hProcess, NULL, 2048,MEM_COMMIT,
PAGE_EXECUTE_READWRITE
        mov    [MemBase],eax
        or     eax,eax
        mov    ebx,offset TextVAE
        jz     ErrMsg
;写入远程序地址空间，把我们的子程序复制过去
        call WriteProcessMemory,PI.hProcess,eax,offset RemoteThreadProc,
RemoteThreadProcLeng, offset NumWritten
        or     eax,eax
        mov    ebx,offset TextWPM
        jz     ErrMsg
;建立远程序线程
        call CreateRemoteThread,PI.hProcess,NULL,0,MemBase,0,0,offset dwThreadNew
        mov    [hThreadNew],eax
        or     eax,eax
        mov    ebx,offset TextCRT
        jz     ErrMsg
        jmp    OK
ErrMsg:
        call MessageBoxA,0,ebx,offset CaptionFail,MB_OK
OK:
        or     hThreadNew,0
        jz     @1
        call CloseHandle,hThreadNew
@1:     or     PI.hProcess,0
        jz     @2
        call CloseHandle,PI.hProcess
@2:     or     PI.hProcess,0
        jz     @3
        call CloseHandle,PI.hThread
```

```
@3: ret
Ingress endp
end WinMain
```

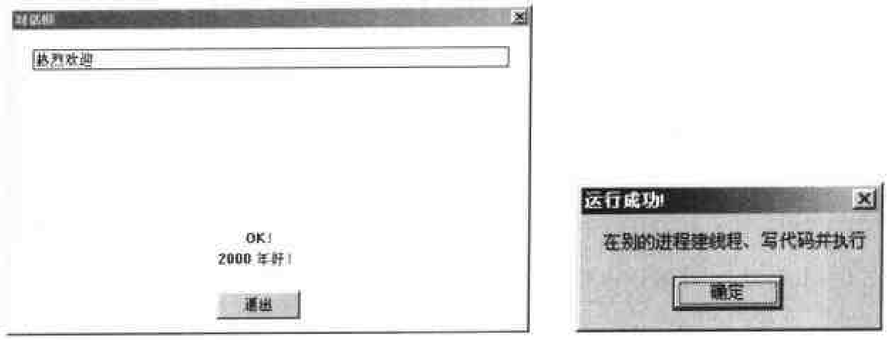
另一个在把代码及数据写入远程进程的方法不需用 VirtualAllocEx，这个 API 在 Windows NT 4.0 之后才提供。我们在调用 CreateRemoteThread 时，系统其实就为我们在远程地址空间中分配了内存——那就是堆栈。在 Win32 中，堆栈、代码、数据在同一平面的地址空间中，堆栈中的代码同样可以执行，我们可以利用这个特征，把代码写入用 CreateRemoteThread 分配的堆栈空间中。

在调用 CreateRemoteThread 时，使用 CREATE_SUSPENDED 标志让线程建立后挂起，而不是立即运行，再用 GetThreadContext 取得线程的寄存器内容，就可以得到 esp 的内容，它指向了线程的堆栈空间。这时，调用函数 VirtualQueryEx 可取得堆栈的内存分配情况。VirtualQueryEx 的用法如下：

```
MEMORY_BASIC_INFORMATION STRUC
    BaseAddress        DD ?
    AllocationBase     DD ?
    AllocationProtect  DD ?
    RegionSize         DD ?
    State      DD ?
    Protect    DD ?
    Type       DD ?
ENDS
    Mbi MEMORY_BASIC_INFORMATION<>
......
call VirtualQueryEx,hProcess,Addr,offset Mbi,size Mbi
```

函数成功时，将内存的情况填入一个 MEMORY_BASIC_INFORMATION 结构中，它里面放有指定内存的基地址、长度、保护情况等，这样我们就可以把代码和数据写入堆栈的基地址，并执行它了。

下面的例子也是建立一个进程（前面的 Asm32.exe），并用 CreateRemoteThread 在它的地址空间中分配一内存，写入代码及数据，然后执行它，结果如图 5-13 所示。



（a）　　　　（b）

图 5-13

```
;*********************************
;文件： WritePr2.asm              *
;功能：写入别的进程的地址空间     *
;*********************************
.386p
.model flat,stdcall

extrn MessageBoxA:proc
extrn ExitProcess:proc
extrn ExitThread:proc
extrn CreateProcessA:proc
extrn CloseHandle:proc
extrn CreateRemoteThread:proc
extrn WriteProcessMemory:proc
extrn ResumeThread:proc
extrn GetLastError:proc
extrn GetThreadContext:proc
extrn SetThreadContext:proc
extrn VirtualQueryEx:proc
extrn GetModuleHandleA:proc
extrn GetProcAddress:proc
extrn GetExitCodeThread:proc
extrn Sleep:proc

MB_OK = 0
NULL   = 0
FALSE = 0
INFINITE     = 0FFFFFFFFH
CREATE_SUSPENDED    = 00000004H
SIZE_OF_80387_REGISTERS      =    80

STATUS_PENDING     = 00000103H
STILL_ACTIVE         = STATUS_PENDING

CONTEXT_i386     = 00010000H
CONTEXT_i486     = 00010000H

CONTEXT_CONTROL = CONTEXT_i386 or 00000001H    ; SS:SP, CS:IP, FLAGS, BP
CONTEXT_INTEGER = CONTEXT_i386 or 00000002H    ; AX, BX, CX, DX, SI, DI
```

```
MEMORY_BASIC_INFORMATION STRUC
    BaseAddress    DD ?            ;base address of region
    AllocationBase      DD ?       ;allocation base address
    AllocationProtect   DD ?       ;initial access protection
    RegionSize     DD ?            ;size, in bytes, of region
    State       DD ?               ;committed, reserved, free
    Protect          DD ?          ;current access protection
    Type       DD ?                ;type of pages
ENDS

STARTUPINFO STRUC
    cb      DD ?
    lpReservedDD ?
    lpDesktop  DD ?
    lpTitle      DD ?
    dwX  DD ?
    dwY  DD ?
    dwXSize   DD ?
    dwYSize   DD ?
    dwXCountChars       DD ?
    dwYCountChars       DD ?
    dwFillAttribute       DD ?
    dwFlags               DD ?
    wShowWindow    DW ?
    cbReserved2     DW ?
    lpReserved2     DD ?
    hStdInput       DD ?
    hStdOutput            DD ?
    hStdError       DD ?
ENDS

PROCESS_INFOMATION STRUC
    hProcess  DD ?
    hThread        DD ?
    dwProcessId   DD ?
    dwThreadId    DD ?
ENDS
FLOATING_SAVE_AREA STRUC
    ControlWord   DD ?
```

```
        StatusWord     DD ?
        TagWord        DD ?
        ErrorOffset    DD ?
        ErrorSelector  DD ?
        DataOffset     DD ?
        DataSelector   DD ?
        RegisterArea       DB SIZE_OF_80387_REGISTERS DUP(?)
        Cr0NpxState    DD ?
    ENDS

    CONTEXT STRUC
        ContextFlags DD ?
        X86Dr0   DD ?
        X86Dr1   DD ?
        X86Dr2   DD ?
        X86Dr3   DD ?
        X86Dr6   DD ?
        X86Dr7   DD ?
        FloatSave FLOATING_SAVE_AREA<>
        SegGs     dd ?
        SegFs     dd ?
        SegEs     dd ?
        SegDs     dd ?
        X86Edi   dd ?
        X86Esi   dd ?
        X86Ebx   dd ?
        X86Edx   dd ?
        X86Ecx   dd ?
        X86Eax   dd ?
        X86Ebp   dd ?
        X86Eip   dd ?
        SegCs     dd ?
        EFlags    dd ?
        X86Esp   dd ?
        SegSs     dd ?
    ENDS
.data
    Ctt   CONTEXT<>
    Mbi MEMORY_BASIC_INFORMATION<>
```

```
    SUI STARTUPINFO<17*4,>
    PI   PROCESS_INFOMATION<>
    ProcessName  db '..\..\Chapter3\Asm32\Asm32.exe',0
    CaptionLoad  db '载入进程',0
    CaptionCRT0  db '建立远程线程(0)',0
    CaptionCRT1  db '建立远程线程(1)',0
    CaptionGTC   db '取线程 CONTEXT:call GetThreadContext',0
    CaptionVQE   db 'call VirtualQueryEx',0
    CaptionWPM   db 'call WriteMemoryProcess',0
    CaptionGMHK      db 'call GetModule--Kernel32',0
    CaptionGMHU      db 'call GetModule--User32',0
    CaptionGAET db 'call GetProcessAddr--ExitThread',0
    CaptionGAMBA     db 'call GetProcessAddr--MessageBoxA',0
    CaptionSTC   db 'call SetThreadContext',0
    TextFail         db '失败!',0
    NumWritten   dd ?
    hKernel          dd ?
    hUser            dd ?
    hGdi     dd ?
    dwThreadNewdd ?
    hThreadNew   dd ?
    dwThreadNew1     dd ?
    hThreadNew1 dd ?
    ThreadExitCode     dd ?
    szKernel  db 'Kernel32.dll',0
    szUser           db 'User32.dll',0
    szGdi            db 'Gdi32.dll',0
    szExitThread  db 'ExitThread',0
    szMessageBoxA     db 'MessageBoxA',0
;=====================================================
OtherProcessProc label byte
    call Addr
Addr:
    pop ebx
    mov eax,ebx
    mov esi,ebx
    mov edi,ebx
    add ebx,TextStr-Addr
    add eax,CaptionStr-Addr
```

```
        add esi,AddrOfMsgBoxA-Addr
        add edi,AddrOfExitThread-Addr
        push edi
        call [esi],NULL,ebx,eax,MB_OK         ;call MessageBoxA
        pop edi
        call [edi],eax   ;CALL ExitThread
        AddrOfMsgBoxA          dd ?
        AddrOfExitThread  dd ?
        TextStr          db '在别的进程建线程、写代码并执行',0
        CaptionStr       db '运行成功!',0
    OtherProcessProcLen=$-OtherProcessProc
    ;+++++++++++++++++++++++++++++++++++++++++++++++++
    .code
    main:
        call CreateProcessA,offset ProcessName,0,0,0,FALSE,0,0,0,offset SUI,offset PI
        or eax,eax
        jnz Next0
        mov eax,offset CaptionLoad
        jmp Error
    Next0:
    ;建立远程线程以取得空间
            call  CreateRemoteThread,  PI.hProcess,NULL,  OtherProcessProcLen+1024,  0,
NULL, CREATE_SUSPENDED, offset dwThreadNew
        mov hThreadNew,eax
        or eax,eax
        jnz Next1
        mov eax,offset CaptionCRT0
        jmp Error
    Next1:
        mov Ctt.ContextFlags,CONTEXT_CONTROL or CONTEXT_INTEGER
        call GetThreadContext,hThreadNew,offset Ctt
        or eax,eax
        jnz Next2
        mov eax,offset CaptionGTC
        jmp Error
    Next2:
        mov ebx,Ctt.X86Esp
        dec ebx
        call VirtualQueryEx,PI.hProcess,ebx,offset Mbi,size Mbi
```

```
        cmp eax,size Mbi
        jz   Next3
        mov eax,offset CaptionVQE
        jmp Error
    Next3:
        call GetModuleHandleA,offset szKernel          ;取 Kernal32.dll 句柄
        mov hKernel,eax
        or eax,eax
        jnz Next31
        mov eax,offset CaptionGMHK
        jmp Error
    Next31:
        call GetModuleHandleA,offset szUser        ;取 User32.dll 句柄
        mov hUser,eax
        or eax,eax
        jnz Next32
        mov eax,offset CaptionGMHU
        jmp Error
    Next32:
        call GetProcAddress,hUser,offset szMessageBoxA
        mov AddrOfMsgBoxA,eax
        or eax,eax
        jnz Next33
        mov eax,offset CaptionGAMBA
        jmp Error
    Next33:
        call GetProcAddress,hKernel,offset szExitThread
        mov AddrOfExitThread,eax
        mov Ctt.X86Eip,eax             ;把线程的开始运行地址放在 ExitThread
        or eax,eax
        jnz Next4
        mov eax,offset CaptionGAET
        jmp Error
    Next4:
    ;把我们的子程序写入远程地址空间
        call WriteProcessMemory, PI.hProcess, Mbi.BaseAddress, offset OtherProcessProc,
OtherProcessProcLen, offset NumWritten
        or eax,eax
        jnz Next5
```

```
        mov eax,offset CaptionWPM
        jmp Error
    Next5:
        call SetThreadContext,hThreadNew,offset Ctt
        or eax,eax
        jnz Next6
        mov eax,offset CaptionSTC
        jmp Error
    Next6:
        ;建立线程并运行
        call  CreateRemoteThread,PI.hProcess,NULL,0,Mbi.BaseAddress,0,0,offset  dwThread
New1
        mov hThreadNew1,eax
        or eax,eax
        jnz Next7
        mov eax,offset CaptionCRT1
        jmp Error
    Next7:
        call Sleep,1500
        call GetExitCodeThread,hThreadNew1,offset ThreadExitCode
        cmp [ThreadExitCode],STILL_ACTIVE
        jz Next7
        call ResumeThread,hThreadNew
        jmp Complete
    Error:
        call MessageBoxA,0,offset TextFail ,eax,MB_OK
    Complete:
        cmp PI.hProcess,0
        jz   Exit
        call CloseHandle,PI.hProcess
        call CloseHandle,PI.hThread
        cmp   hThreadNew,0
        jz Exit
        call CloseHandle,hThreadNew
        cmp hThreadNew1,0
        jz Exit
        call CloseHandle,hThreadNew1
    Exit:call ExitProcess,0
    end main
```

5.6 钩子过程

一个线程一般来说只应得到它自己的消息，但并不总是如此，也可能想要收到整个系统的某一特定消息，这时可以使用钩子过程。

当说到“拦截键盘、鼠标消息”时，DOS 程序员会很容易想到修改中断向量表，但是这种方法不太适用于 Windows。在 Windows 95/98 中，确实可以修改中断描述符表，但是，那样不但麻烦，而且中断程序必须在特权级 0 上运行，难以调试又易出问题，而大多数 Win32 API 都不能在特权级 0 上使用，更何况在 Windows NT 中这个方法根本行不通。

Windows 提供 SetWindowsHookEx 函数来提供对消息挂钩，即把我们定义的函数挂在一个消息处理的过程链上。SetWindowsHookEx 可以只监视调用线程自身所在进程的某一线程的消息，也可以是系统中所有的消息。当要监视系统中所有线程的消息时，必须把钩子过程放在一个动态链接库中。

下面是安装钩子过程的方法：

```
HookProc proc nCode:DWORD,wParam:DWORD,lParam:DWORD
;钩子处理
HookProc endp
call SetWindowsHookEx,idHook,offset HookProc,hMod,threadID
mov hHook,eax
```

其中，idHook 是安装的钩子类型。它有很多值可以选，如对鼠标消息感兴趣，可用 WH_MOUSE；要在消息处理函数处理后取得控制，可以用 WH_CALLWNDPROCRET 等等。在下面所举的例子中，我们可以使用值 WH_JOURNALRECORD 来监视系统中所有的键盘消息。消息处理函数要放在一个 DLL 文件中。

如果在系统中挂上过多的钩子会影响系统的速度，因此，当不需要时应尽快删除它：

```
call UnhookWindowsHookEx,[hHook]
```

如图 5-14 所示的例子中，程序初始化时加载 DLL，取得 DLL 中提供的安装钩子过程的函数地址，并调用它。钩子过程在消息到来时，检查是否是键盘消息，是则发消息告诉我们的主窗体，然后调用 CallNextHookEx 把控制传递给下一个挂钩过程。

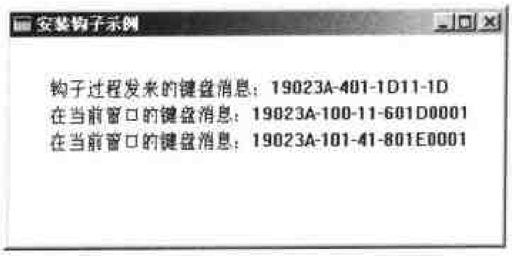


图 5-14

在主程序中，当接到键盘消息或钩子过程发送过来的消息时，则将消息处理函数的参数打印出来。

```
;*****************************
;文件：SetHook.asm           *
;功能：设置钩子过程          *
```

```
;****************************

.386p
.model flat,stdcall
include win32.inc

extrn GetMessageA:proc
extrn TranslateMessage:proc
extrn DispatchMessageA:proc
extrn ShowWindow:proc
extrn UpdateWindow:proc
extrn CreateWindowExA:proc
extrn RegisterClassA:proc
extrn DefWindowProcA:proc
extrn MessageBoxA:proc
extrn ExitProcess:proc
extrn LoadIconA:proc
extrn LoadCursorA:proc
extrn GetStockObject:proc
extrn GetModuleHandleA:proc
extrn PostQuitMessage:proc
extrn GetDC:proc
extrn ReleaseDC:proc
extrn TextOutA:proc
extrn LoadLibraryA:proc
extrn GetProcAddress:proc
extrn _wsprintfA:proc

WM_MYMSG = WM_USER + 1          ;自定义消息

POINT struc
    x dd ?
    y dd ?
    ends
MSG struc
    hwnd      dd ?
    message   dd ?
    wParam    dd ?
    lParam    dd ?
```

```
    time dd ?
    pt        POINT<>
    ends

WNDCLASS struc
    style       dd ?
    lpfnWndProc  dd ?
    cbClsExtra    dd ?
    cbWndExtra   dd ?
    hInstance dd ?
    hIcon         dd ?
    hCursor       dd ?
    hbrBackground     dd ?
    lpszMenuName      dd ?
    lpszClassName     dd ?
    ends

.data
    DllFileName  db '..\HookDLL\HookDLL.dll',0
    ProcName     db 'InstallKeyHook',0
    PintMsg_Self db '在当前窗口的键盘消息：%lX-%lX-%lX-%lX                ',0
    PintMsg_Dll        db '钩子过程发来的键盘消息：%lX-%lX-%lX-%lX          ',0
    PintMsg      db 100 dup (0)
    myclassname  db 'MyClass',0
    caption      db '安装钩子示例',0
    align 4
    hInstdd ?
    hWnd      dd ?
    msg       MSG<>
    wc        WNDCLASS<>

.code
main:
    call      GetModuleHandleA,0            ;取模块句柄
    mov       [hInst],eax

    mov       wc.lpszClassName,offset myclassname
    mov       eax,[hInst]
    mov       wc.hInstance,eax
```

```
        mov     wc.lpfnWndProc,offset WinMsgProc
        call    LoadIconA,0,IDI_APPLICATION
        mov     wc.hIcon,eax
        call    LoadCursorA,0,IDC_IBEAM
        mov     wc.hCursor,eax
        mov     wc.lpszMenuName,0
        call    GetStockObject,WHITE_BRUSH
        mov     wc.hbrBackground,eax
        mov     wc.style,CS_HREDRAW or CS_VREDRAW or CS_GLOBALCLASS
        mov     wc.cbClsExtra,0
        mov     wc.cbWndExtra,0
        call    RegisterClassA,offset wc     ;注册窗口类

        ;建立主窗口
        call CreateWindowExA, 0, offset myclassname, offset caption,
WS_OVERLAPPEDWINDOW, 100, 100,450,300,0,0,[hInst],0
            mov     [hWnd],eax

        call    ShowWindow,[hWnd],1
        call    UpdateWindow,[hWnd]
    MsgLoop:    ;消息循环
        call    GetMessageA,offset msg,0,0,0
        or      eax,eax
        jz      Exit
        call    TranslateMessage,offset msg
        call    DispatchMessageA,offset msg
        jmp     MsgLoop
    Exit:       ;退出进程
        call    ExitProcess,0
;**********************************************************************
    ;消息处理函数
    WinMsgProc    proc uses ebx edi esi, handle:DWORD, wMsg:DWORD,
wParam1:DWORD, lParam1:DWORD
        LOCAL  hldc:DWORD
            cmp     wMsg,WM_CREATE
        jz      create
            cmp     wMsg,WM_DESTROY
        jz      destory
        cmp     wMsg,WM_KEYDOWN
```

```
        jz      keydown
        cmp     wMsg,WM_KEYUP
        jz      keyup
        cmp     wMsg,WM_MYMSG
        jz      mymsg
        jmp     default
create:         ;建立
        call  LoadLibraryA,offset DllFileName          ;把 HookDLL.dll 映射入进程空间
        call  GetProcAddress,eax,offset ProcName       ;取安装钩子的函数
        call  eax,handle                               ;安装钩子
        xor     eax,eax
        ret
destory:        ;注消窗体
        call    PostQuitMessage,0
        xor     eax,eax
        ret
keydown:        ;有键按下
call    GetDC,handle
mov     hldc,eax
call _  wsprintfA,offset PintMsg,offset PintMsg_Self,handle,wMsg,wParam1,lParam1
add     esp,4*6
call    TextOutA,hldc,30,50,offset PintMsg,eax
call    ReleaseDC,handle,hldc
xor     eax,eax
ret
keyup:                                  ;有键放开
        call  GetDC,handle
        mov     hldc,eax
        call  _wsprintfA,offset PintMsg,offset PintMsg_Self,handle,wMsg,wParam1,lParam1
        add     esp,4*6
        call  TextOutA,hldc,30,70,offset PintMsg,eax
        call  ReleaseDC,handle,hldc
        xor     eax,eax
        ret
mymsg:                                  ;DLL 发消息来，有键按下
        call  GetDC,handle
        mov     hldc,eax
        call  _wsprintfA,offset PintMsg,offset PintMsg_Dll,handle,wMsg,wParam1,lParam1
        add     esp,4*6
```

```
       call  TextOutA,hldc,30,30,offset PintMsg,eax
       call  ReleaseDC,handle,hldc
             xor        eax,eax
       ret
default:
             call  DefWindowProcA,handle,wMsg,wParam1,lParam1
       ret
WinMsgProc   endp
end main
```

以下是动态链接库的源程序，因为钩子过程必须放在 DLL 中才可以钩住所有的过程。

```
;*****************************
;文件：HookDLL.asm           *
;功能：在 DLL 中的钩子过程    *
;*****************************
.386P
.Model Flat ,StdCall
include win32.inc
extrn MessageBoxA:PROC
extrn SetWindowsHookEx:PROC
extrn CallNextHookEx:PROC
extrn SendMessageA:PROC
extrn PostMessageA:PROC
extrn SetWindowsHookExA:PROC
extrn UnhookWindowsHookEx:PROC

NULL = 0
WM_MYMSG = WM_USER + 1        ;自定义消息

DLL_PROCESS_ATTACH = 1
DLL_PROCESS_DETACH = 0
DLL_THREAD_ATTACH   = 2
DLL_THREAD_DETACH   = 3

HC_ACTION = 0
WH_JOURNALRECORD = 0

EVENTMSG struc
     message  dd ?
     paramL   dd ?
```

```
        paramH   dd ?
        time dd ?
        handle     dd ?
        ends

    public KeyHook
    public InstallKeyHook

    .Data
        Caption   db 'Hook DLL',0
        pinit db 'DLL 装入',0
        penddb 'DLL 退出',0

        hMainWin       dd ?               ;主窗口句柄
        hModule        dd ?               ;DLL 的模块句柄
        bHook          dd ?               ;钩子句柄

    .Code
    DllMain proc hInst:DWORD,fdwReason:DWORD,fImpLoad:DWORD
        mov      eax,fdwReason
        cmp      eax,DLL_PROCESS_ATTACH
        jz       ProcInit
        cmp      eax,DLL_PROCESS_DETACH
        jz       ProcExit
        cmp      eax,DLL_THREAD_ATTACH
        jz       ThreadInit
        cmp      eax,DLL_THREAD_DETACH
        jz       ThreadExit
    ProcInit:                          ;DLL 初映射进进程空间
        mov      eax,hInst
        mov      [hModule],eax
        call  MessageBoxA,NULL,offset pinit,offset Caption,MB_OK
        jmp      DllMainEnd
    ProcExit:                          ;解除映射
        call  MessageBoxA,NULL,offset pend,offset Caption,MB_OK
        call  UnhookWindowsHookEx,dword ptr [hHook]
        jmp      DllMainEnd
    ThreadInit:
        jmp      DllMainEnd
```

```
ThreadExit:
    jmp     DllMainEnd

DllMainEnd:
    mov     eax,1
    ret
DllMain endp
;*************************************************
KeyHook proc nCode:DWORD,wParam:DWORD,lParam:DWORD
    cmp     nCode,HC_ACTION
    jnz     KeyHookEnd
    mov     edx,lParam
    mov     eax,[edx]
    cmp     eax,WM_KEYDOWN          ;是键盘按下?
    jnz     KeyHookEnd

    mov     eax,[edx].paramL        ;送信息给主窗口
    mov     edx,[edx].paramH
    call  PostMessageA,dword ptr [hMainWin],WM_MYMSG,eax,edx

KeyHookEnd:                         ; 下一个钩子过程
    call  CallNextHookEx,hHook,nCode,wParam,lParam
    ret
KeyHook Endp

InstallKeyHook proc hwin:DWORD      ;安装钩子过程
    mov     eax,hwin
    mov     [hMainWin],eax
    call  SetWindowsHookExA,WH_JOURNALRECORD,offset    KeyHook,dword    ptr
[hModule],0
    mov     [hHook],eax
    ret
InstallKeyHook endp
end DllMain
```

第六章　中断与 I/O 操作

本章主要介绍中断与输入输出程序的编写。包括 80X86 的中断与异常的理论，结构化异常处理的过程，硬件中断的映射，给软中断挂钩，虚拟化 IRQ，I/O 操作等。

6.1　80X86 的中断与异常

从学习 8088/8086 汇编语言我们知道，中断可以被分成 2 类：内部中断和外部中断。在 80386 中，把外部中断称为“中断”，而内部中断则称为“异常”。以后我一般会分开这两个术语，但有时为了文字简洁，在不至于混淆的情况下会统称为“中断”。

中断是由外部硬件产生，它可分为可屏蔽中断和不可屏蔽中断。可屏蔽中断直接引至 CPU 的引脚 NMI ，它的中断号固定为 2；而不可屏蔽中断则通过可编程中断控制器 8259A 或其兼容芯片引到 CPU 引脚 INTR，它们的中断号可以通过对 8259A 编程改变。

异常与执行的指令有关，它又可以分为 3 类：

1）故障：故障所保存的断点 CS：EIP 是引起故障的指令，即从故障处理程序返回时，应该再执行引起故障的指令。例如段不存在、页故障等。

2）陷阱：陷阱所保存的断点 CS：EIP 是执行引起异常的指令之后下一条要执行的指令。注意是下一条要执行的指令而不是下一条指令。这种异常有软件中断、单步异常等。

3）中止：中止是被认为是非常严重的异常，它不能恢复执行。

某些异常还会给出一个 16 位的出错码，压在堆栈中，用以说明出错的原因。这里不细说出错码的格式了，但要注意实际设计系统时在中断返回前要调整堆栈，丢弃掉出错码（我们下面的例子没有处理这种情况），如表 6-1 所示是异常的列表。

表 6-1　异常列表

NO	描述	类型	有否出错码	引起异常的指令或原因
0	除以 0	故障	无	DIV，IDIV
1	单步	故障/异常	无	标志寄存器中的单步位
3	Int 3	陷阱	无	INT 3
4	溢出	陷阱	无	INTO
5	边界检查	故障	无	BOUND
6	错误的操作码	故障	无	指令编码错
7	设备不可用	故障	无	浮点或 wait 指令
8	双重故障	中止	有	异常自理中发生异常
9	协处理器段越界	中止	无	访问存储器的浮点指令
A	无效 TSS	故障	有	JMP、CALL、IRET、中断
B	段不存在	故障	有	装载段寄存器
C	堆栈段异常	故障	有	装载 SS 寄存器或 SS 段寻址
D	通用保护	故障	有	特权指令、访问存储器指令

续表 6-1

NO	描述	类型	有否出错码	引起异常的指令或原因
E	页异常	故障	有	访问存储器
10	协处理器出错	故障	无	浮点或 wait 指令
0~FF	Intn 指令	陷阱	无	INTN

这时，你大概对 Windows 9X 的蓝屏信息如“致命异常 0E 发生在 XXXX：XXXXXXXX”之类的有一些了解了吧？

在 8086/8088 中，当发生中断时，CPU 到以物理地址 0 为基地址的“中断向量表”取得中断处理程序的地址，每个中断地址占 4 字节——CS 和 IP 各占 2 字节。在 80386 中，“中断向量表”被称为“中断描述符表”，它不再是一定处于物理地址 0，而是由一个“中断描述符表寄存器”IDTR 指出它的基地址，它的第一个中断入口也不再是 4 字节，而是变成了 8 字节，如图 6-1 所示。

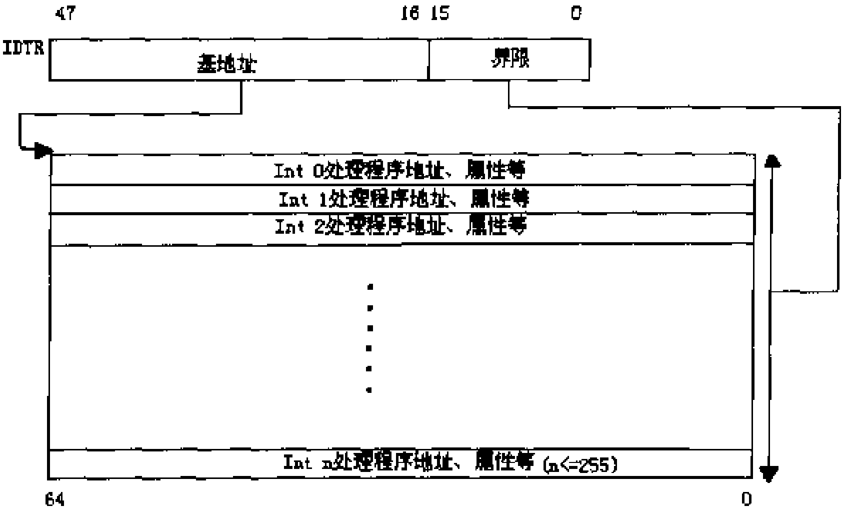


图 6-1

下面的例子演示了如何建立中断描述符表及中断处理过程。程序用一个 REPT 宏生成中断描述符表。中断处理程序的开始也用 REPT 宏来生成，它先保存好中断向量号，然后跳往一个共同的处理过程，它的功能是显示如下信息：

Interrupt Example: Int xx

其中 xx 是中断号。

程序一开始先建立各种表格，并进入保护模式，然后用 Int 指令产生中断，显示上面的信息，然后重新设置 IDTR，使其指向物理地址 0。之所以要重新设置 IDTR，是因为 IDTR 并不仅仅在保护模式下才有用。也就是说，你在实模式下也可以通过设置 IDTR 而使得系统认为中断向量表不在物理地址 0。

```
;******************************
;文件：X86Int.asm            *
;功能：中断/异常处理例子      *
;******************************
```

```
.386p
TestIntNum equ 3EH
JUMP16 MACRO selector,offsetv
      DB 0EAH
      DW offsetv
      DW selector
      ENDM
;---------------------
Descriptor STRUC
      Limitl    dw 0
      basel     dw 0
      basem     db 0
      attrib    dw 0
      baseh      db 0
Descriptor ENDS
;---------------------
Gate STRUC
      offsetl    dw 0
      selector dw 0
      dcount     db 0
      gtype        db 0
      offseth    dw 0
Gate ENDS
;==============================
Data Segment use16
      gdt0 Descriptor <>

      DataSel = $-gdt0
      DataDes Descriptor <0ffffh,,,92H,>

      CodeSel = $-gdt0
      CodeDes Descriptor <0ffffh,,,98H,>

      VideoSel = $-gdt0
      VideoDes Descriptor <0ffffh,8000H,0BH,92H,0>

      Stack0Sel = $-gdt0
      Stack0Des Descriptor <0,,,97H,>
```

```
        IntProcSel = $-gdt0
        IntProcDes Descriptor <0ffffh,,,98H,>

        GdtLen   = $-gdt0

        GdtPtr   dw GdtLen-1
             dd 0
;+++++++++++++++++++++++++++++++++++++++++++++
  IdtTable label byte
  Off=offset IntProcBegin
      REPT 256
           Gate <Off,IntProcSel,0,0eeH,0>
           Off=Off+9
      ENDM
      IdtLen = $-IdtTable
      IdtPtr dw IdtLen-1
          dd 0
      OldIdt dw 0
          dd 0
;+++++++++++++++++++++++++++++++++++++++++++++
OldSPSS dw 0,0
      Mess db 'Interrupt Example: Int '
         M1 db '   ',0
    Digit db '0123456789ABCDEF'
Data ends
;=============================================
Stack0 Segment use16
      db 1024 dup (0)
Stack0 ends
;=============================================
IntProc Segment use16        ;中断处理程序
      assume cs:IntProc
IntProcBegin:
      COUNT = 0
      REPT 256
      push word ptr COUNT
      push ax
      mov ax,offset All
```

```
        jmp ax
        COUNT = COUNT+1
            ENDM
    All:
        pop ax
        push bp
        mov bp,sp
        pusha
        push ds
        push es
        mov ax,DataSel
        mov ds,ax
        mov ax,VideoSel
        mov es,ax
        mov ax,[bp+2]
        and ax,0FFH
        push ax
        mov bx,offset Digit
        mov si,offset M1
        and ax,0fh
        xlat
        mov [si+1],al
        pop ax
        shr al,4
        xlat
        mov [si],al

        mov si,offset Mess  ;显示"Interrupt Example: Int xx
        mov di,80*2*10
        cld
        mov ah,1eh
Load:
        lodsb
        cmp al,0
        jz OK
        stosw
        jmp Load
OK:
```

```
        pop es
        pop ds
        popa
        pop bp
        add sp,2
        iretd
IntProc ends
;==========================================
Code segment use16
        assume cs:code,ds:data
Start:
        xor eax,eax             ;Gdt
        mov ax,Data
        mov ds,ax
        shl eax,4
        mov dword ptr [GdtPtr+2],eax
        xor eax,eax             ;初始化 Idt
        mov ax,Data
        shl eax,4
        add eax,offset IdtTable
        mov dword ptr [IdtPtr+2],eax
        xor eax,eax             ;初始化数据段描述符
        mov ax,Data
        shl eax,4
        mov DataDes.basel,ax
        shr eax,16
        mov DataDes.basem,al
        mov DataDes.baseh,ah
        xor eax,eax             ;初始化代码段描述符
        mov ax,Code
        shl eax,4
        mov CodeDes.basel,ax
        shr eax,16
        mov CodeDes.basem,al
        mov CodeDes.baseh,ah
        xor eax,eax             ;初始化堆栈段描述符
        mov ax,Stack0
        shl eax,4
```

```
        mov Stack0Des.basel,ax
        shr eax,16
        mov Stack0Des.basem,al
        mov Stack0Des.baseh,ah
        xor eax,eax             ;初始化中断处理段的描述符
        mov ax,IntProc
        shl eax,4
        mov IntProcDes.basel,ax
        shr eax,16
        mov IntProcDes.basem,al
        mov IntProcDes.baseh,ah

        lgdt qword ptr GdtPtr   ;载入 GDTR
        cli
        sidt qword ptr OldIdt   ;保存 IDTR
        lidt qword ptr IdtPtr   ;载入 IDTR
        mov OldSPSS,sp
        mov [OldSPSS+2],ss
        mov eax,cr0             ;转入保护模式
        or eax,1
        mov cr0,eax
        JUMP16 CodeSEL,<offset Protect>
Protect:
        mov ax,DataSel
        mov ds,ax
        mov fs,ax
        mov gs,ax
        mov ax,Stack0Sel
        mov ss,ax
        mov sp,1024
        mov ax,VideoSel
        mov es,ax

        int TestIntNum          产生中断
ReadyToReal:                    ;反回实模式
        mov eax,cr0
        and eax,0fffffffeh
        mov cr0,eax
```

```
        JUMP16 <seg Real>,<offset Real>
    Real:
        mov ax,Data
        mov ds,ax
        mov es,ax
        lss sp,dword ptr OldSPSS
        lidt qword ptr OldIdt
        sti
        mov ax,4c00h
        int 21h
    Code ends
    end Start
```

6.2 结构化异常处理

当某一线程发生异常时，程序的控制权会立即进入 Ring0 异常处理程序，这是属于操作系统的部分，如果发生的异常是如页异常之类的异常，Ring0 处理程序可以处理完它后重新回到程序中执行，而被中断过和进程可能根本就不知道发生过异常。

但事情并不总是如此，有时进程会发生一些始料不及的异常，例如访问不存在的内存，被 0 除等，这些异常 Ring0 处理程序不知该如何处理它，而进程本身也可能想自己处理这些情况，这时就要用到结构化异常处理（SHE）。在 C/C++中也有异常处理的语句如 _try,_catch 等，这些语句的实现也与 SEH 紧密联系。

当系统遇到一个它不知如何处理的异常时，它就查找异常处理链表，注意每个线程都有它自己的异常处理链表。异常链表以 FS:[0]所指向的位置为 FS:[0] = RRRRRRRRH 链表头，如图 6-2 所示。

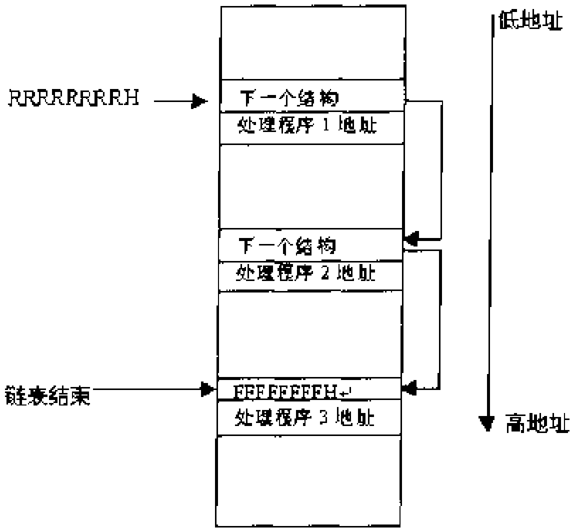


图 6-2

异常处理开始时，系统把一些与当前线程和与异常有关的内容传给链头所指向的处理程序；处理程序由用户编写或编译器生成，它的返回值可以是告诉系统：异常处理已完成，可以继续执行程序，或未处理异常，可由链表的下一个处理程序处理等，可以依次传递下去。

下面的程序演示了一个结构异常处理的例子。这个例子非常的简单，异常处理只是简单地显示一个对话框，然后退出进程。

程序开始先在异常处理链表头插入我们自己的异常处理程序，然后通过读线性地址 0 来产生异常，进入异常处理程序。

例程 1：

```
;**********************************
;文件：SEH.ASM                    *
;功能：演示结构化异常处理         *
;**********************************
.386p
locals
.model flat,stdcall
include win32.inc
extrn MessageBoxA:proc
extrn ExitProcess:proc

.data
    Caption db 'SEH',0
    TextSEH  db 'SEH 程序正在运行',0
    Text db 'SEH 程序没有运行',0

.code
main:
    lea    eax,[esp-4*2]
    xchg   fs:[0],eax
    mov    ebx,offset SEH
    push   ebx
    push   eax

    mov    esi,0
    mov    eax,[esi]             ;读 NULL 指针，产生异常
    call   MessageBoxA,0,offset Text,offset Caption,MB_OK
    jmp    Exit
SEH:
```

```
        call    MessageBoxA,0,offset TextSEH,offset Caption,MB_OK
    Exit:
        call    ExitProcess,0
    end main
```

左前面章节，我们在谈及线程的堆栈时曾说过，保留堆栈地址的分配是通过存取边界的保护页从而使系统分配一页的。但是，当程序存取堆栈时，若跨越了保护页，则会产生异常。我们这里写两个程序，它们写的页都跨越了已提交的可读可写页、已提交的保护页和未提交的页，但它们写的顺序不同：StackNormal.exe 是从高位向低位写，因此可以顺利通过；而 StackOver.exe 是从低位向高位写，因而产生内存访问异常，会被我们的结构化异常处理程序截到。

如图 6-3 所示是两个程序的运行结果。若程序正常结束，则弹出对话框显示“Normal”，否则在结构化异常处理程序中弹出对话框，显示“Overflow”。

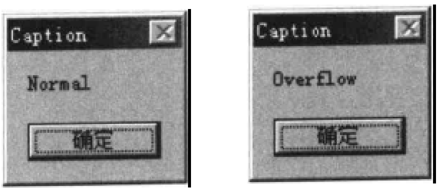


图 6-3

程序在 Windows 98 中调试通过。

例程 2:

下面是 StackNormal 的源程序，它可以正常写完堆栈。

```
;********************************
;文件：StackNormal.asm          *
;功能：与 StackOver 一起测试在   *
;       堆栈越界时的页分配的过程*
;*********************************

.386p
.model flat,stdcall
NULL   =0
MB_OK = 0
COMMIT   = 2000H
RESERVE = 100000H
extrn MessageBoxA:proc
extrn ExitProcess:proc

.data
    Caption db 'Caption',0
```

```
    TextNormal     db 'Normal',0
    TextOver db 'Overflow',0
    EspTemp dd ?
.code
main:
    lea     eax,[esp-4*2]           ;生成异常处理链表
    xchg    fs:[0],eax
    mov     ebx,offset SEH
    push    ebx
    push    eax
    mov [EspTemp],esp
    call WriteStack

    call MessageBoxA,NULL,offset TextNormal,offset Caption,MB_OK

    pop eax
    pop ebx
    xchg fs:[0],eax
    call ExitProcess,0
SEH:
    mov esp,[EspTemp]
    call MessageBoxA,NULL,offset TextOver,offset Caption,MB_OK
    call ExitProcess,0

WriteStack proc                ;写堆栈
    push ebp
    mov ebp,esp

    lea edi,[esp-1]
    mov ecx,COMMIT + 4096
    mov al,0
    std
    rep stosb

    mov esp,ebp
    pop ebp
    ret
WriteStack endp
```

```
end main
```

例程 3:

下面是 StackNormal 的源程序，它会产生堆栈溢出。

```
;************************************
;文件：StackOver.asm                 *
;功能：与 StackNormal 一起测试在      *
;           堆栈越界时的页分配的过程*
;************************************

……；同上

WriteStack proc                 ;写堆栈
      push ebp
      mov ebp,esp

      mov ecx,COMMIT + 4096
      lea edi,[esp-1]
      sub edi,ecx
      mov al,0
      cld
      rep stosb

      mov esp,ebp
      pop ebp
      ret
WriteStack endp

end main
```

6.3 硬件中断的映射

典型的 PC 机使用 8259A 作为中断控制器，它把来自硬件的中断送给 CPU，并告知 CPU 中断号，从而使 CPU 知道到底中断来自哪个硬件。现在的 PC 机虽然速度比以前的机器快许多倍，中断控制器也很多都采用了更先进的芯片，如 82380 等，但仍保持程序的兼容。

在 DOS 下写过如热键激活、时钟等 TSR 程序的汇编程序员大概都知道，键盘的硬件中断号是 9，定时器是 8 等等，这些都是由 8259A 或其兼容芯片把 IRQ 上的信号转换成相应的中断号传给 CPU，然后 CPU 进行相应的处理。

其映射关系一般如表 6-2 所示。

表 6-2 IRQ 与中断号之间的映射关系

IRQ	中断号	描述
IRQ0	8	定时器
IRQ1	9	键盘
IRQ2	A	重定向至 8259A 从片
IRQ3	B	COM2
IRQ4	C	COM1
IRQ5	D	LPT2
IRQ6	E	软盘
IRQ7	F	LPT1

IRQ 与硬件的对应关系，其实可以在 Windows 中查出。例如，在 Windows 98 中，右键单击桌面上的“我的电脑”，选“属性”，然后选“设备管理器”页，双击“计算机”，即可看到其对应关系，如图 6-4 所示。

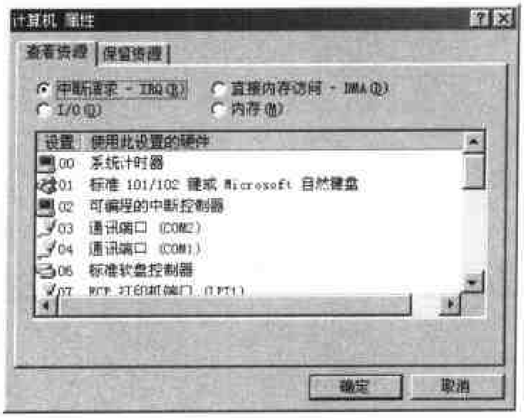


图 6-4

但在表 6-2 中那种 IRQ 与中断号之间的映射关系并不是固定的，8259A 被称作“可编程中断控制器”，这种映射是可通过对 8259A 编程来改变。在 Windows 9X 中，IRQ0 被映射到中断 50H，IRQ1 映射到中断 51H……，依次类推，即可得其映射表 6-3；而 IRQ8~F 则映射到时中断 58H~5FH。

表 6-3 Windows 9X 下的中断映射

IRQ	中断号	描述
IRQ0	50H	定时器
IRQ1	51H	键盘
IRQ2	52H	重定向至 8259A 从片
IRQ3	53H	COM2
IRQ4	54H	COM1
IRQ5	55H	LPT2
IRQ6	56H	软盘
IRQ7	57H	LPT1

在 Windows NT 中这种映射又不同，Windows NT 把 IRQ0~7 映射到中断 30H 到 37H，而把 IRQ8~F 映射到中断 38H~3FH。

这样，在 Windows 9X 下，若您想改变定时器中断，就要改变中断 50H；要改键盘中断，就要改中断 51H，等等。下面我们编写一个例程，改变键盘中断，使它能接到用户的每次击键消息，而不管击键是否在当前进程的窗体中。

首先要说明一点是：要取得所有的击键消息并不需要使用这种技术，您可以用 Win32 API 函数 SetWindowsHookEx 来实现，这里用这种方法仅为演示之用。

程序的界面如图 6-5 所示，程序用一个变量做计数器，每次发生一次键盘中断，这个计数器都有加一。要注意的是：每次击键都会引起两次中断，这是因为击键的压下和松开都会产生一次中断。单击“刷新”会得到最新的击键次数和最后一次中断所取得的键盘扫描码。Hook 地址是新的中断地址。

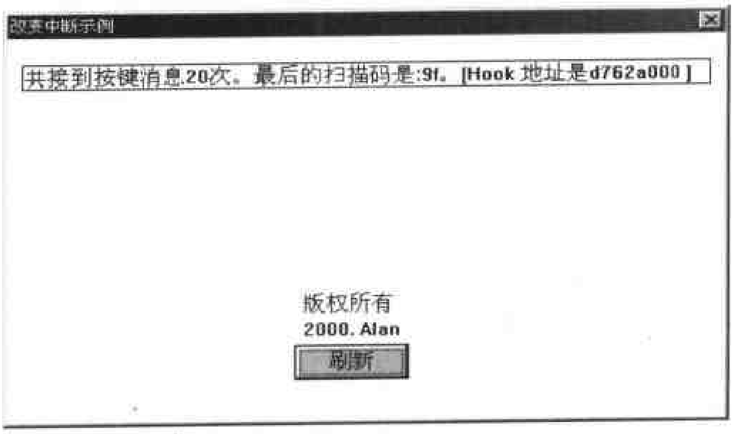


图 6-5

我们知道在 Windows 9X 中可以直接更改中断描述符表，在这个程序中正是使用这种方法。我们在 Ring0 中先分配一块内存，把我们的中断处理程序 Copy 过去，然后把 IDT 中的 51H 中断入口指向我们的处理程序。

下面是源程序，至于.rc 文件，因为跟前面的示例程序 Asm32 差不多，我们就不列出来了。

```
;**********************************
;文件：int.asm                    *
;功能：演示如何在 Windows 9X 下改*
;               变 IDT 表          *
;**********************************
IDC_TIMES   = 100
HookInt = 51H
.386p
locals
.model flat,stdcall
include win32.inc
```

```
extrn MessageBoxA:proc
extrn ExitProcess:proc
extrn DialogBoxParamA:proc
extrn EndDialog:Proc
extrn GetDlgItemTextA:proc
extrn SetDlgItemTextA:proc
extrn GetModuleHandleA:proc
extrn _wsprintfA:PROC
.data
    DialogName  db 'MyDialog',0
    CaptionFail db '失败！',0
    TextFail db 'API:DialogBoxParamA 调用失败！',0
    Addr        dd ?
    CallAddr dd 0
             dw 07h
    Temp        dd  ?,?
    Addr        dd ?
    TIMES   db 100 dup ('A'),0
    Param   db '共接到按键消息%d 次。最后的扫描码是:%x。[Hook 地址是%x ]',0

.code
WinMain:
;**********************************
    push offset InstallHook          ;改变 IDT 表
    call ToRing0Code
    mov [Addr],eax
;**********************************
    call GetModuleHandleA,0
    call DialogBoxParamA,eax,offset DialogName,0,offset DialogProc,0
    cmp eax,0
    jnz  Exit
    mov eax,offset TextFail
    call MessageBoxA,0,eax,offset CaptionFail,MB_OK
Exit:
;**********************************
    push offset UnInstallHook        ;把 IDT 表改回原来的值
    call ToRing0Code
    call ExitProcess,0
```

```
;***********************************
;主窗口消息处理
DialogProc  proc uses ebx edi esi, hWnd:DWORD, wMsg:DWORD, wParam:DWORD,
lParam:DWORD
        cmp wMsg,WM_INITDIALOG
        jz WmInitDialog
        cmp wMsg,WM_CLOSE
        jz WmClose
        cmp wMsg,WM_COMMAND
        jz WmCommand
        jmp ExitProc
    WmInitDialog:
        call _wsprintfA,offset TIMES,offset Param,0,0,[Addr]
        add esp,20
        call SetDlgItemTextA,hWnd,IDC_TIMES,offset TIMES
        jmp ExitProc
    WmClose:
        call EndDialog,hWnd,1
        jmp ExitProc
    WmCommand:
        mov ebx,wParam
        cmp bx,IDOK              ;按了"刷新"按钮?
        jnz ExitProc
        mov ebx,[Addr]
        lea esi,[ebx+Begin-IntProc-8]
        lea edi,[ebx+Begin-IntProc-4]
        call _wsprintfA,offset TIMES,offset Param,dword ptr[esi],dword ptr[edi],[Addr]
        add esp,20
        call SetDlgItemTextA,hWnd,IDC_TIMES,offset TIMES
        jmp ExitProc
    ExitProc:
        xor eax,eax
        ret
    DialogProc endp
;*******************************************************************
ToRing0Code:
        push ebp
        mov ebp,esp
```

```
    call GetLdtAddress
    mov ecx,[eax]           ;Save old Descriptor
    mov [Temp],ecx
    mov ecx,[eax+4]
    mov [Temp+4],ecx
    mov edx,[ebp+8]
    mov [eax],dx
    mov word ptr [eax+2],28h
    mov word ptr [eax+4],0ec00h
    shr edx,16
    mov [eax+6],dx
    push eax
    call fword ptr[CallAddr] ;call Ring0 program
    pop ebx
    mov   edx,[Temp]        ;Load old descriptor
    mov   [ebx],edx
    mov   edx,[Temp+4]
    mov   [ebx+4],edx
    pop ebp
    ret   4

UnInstallHook:
    cli
    call GetHookIntAddrInIdt

    push eax
    mov ebx,[eax+4]
    mov bx,[eax]
    push 1
    push ebx
    int 20H
    dd   10055H             ;VxdCall __PageFree
    add esp,2*4
    pop eax

    mov ebx,OldIntGate
    mov [eax],ebx
    mov ebx,[OldIntGate+4]
```

```
    mov [eax+4],ebx
    sti
    retf

InstallHook:
    cli
    xor eax,eax
    push dword ptr 0fH
    push eax
    push dword ptr 0ffffffffH
    push eax
    push eax
    push eax
    push dword ptr 1
    push dword ptr 2
    int 20H                 ;call _pageAllocate
    dd 10053H
    add esp,8*4

    push eax                ;Save New Interrupt Address
    mov edi,eax
    mov esi,offset IntProc
    mov ecx,OldIntRelativeAddress-IntProc
    cld
    rep movsb
    push ebx
    sidt [esp-2]
    pop ebx
    add ebx,HookInt*8
    mov eax,[ebx+4]
    mov ax,[ebx]
    sub eax,edi
    sub eax,4
    stosd

    call GetHookIntAddrInIdt
    mov ebx,eax
    mov eax,[ebx]
```

```
        mov OldIntGate,eax
        mov eax,[ebx+4]
        mov [OldIntGate+4],eax
        pop eax              ;Load New Interrupt Process address
        mov [ebx],ax
        ror eax,16
        mov [ebx+6],ax
        ror eax,16
        sti
        retf

GetLdtAddress:                ;取 LDT 地址
        push ebx
        sgdt [esp-2]
        pop   ebx
        sldt ax
        and eax,0fff8H
        add   ebx,eax
        mov   eax,[ebx+2]
        mov dl,[ebx+7]
        shl edx,24
        and eax,0ffffffh
        or   eax,edx
        ret
GetHookIntAddrInIdt:
        push eax
        sidt [esp-2]
        pop eax
        add eax,HookInt*8
        ret

IntProc:                      ;中断处理
        pushf
        push eax
        push ebx
        push ds

        mov bx,30H
```

```
    mov ds,bx
    call Begin
    dd 0
    dd 0
Begin:
    pop ebx
    inc dword ptr[ebx]        ;按中断次数加一
    in al,60H
    mov byte ptr[ebx+4],al    ;记下扫描码

    pop ds
    pop ebx
    pop eax
    popf
    db 0e9H
OldIntRelativeAddress:        ;跳回原中断处理程序处
    db 4 dup (?)
;*****************************************************************
end WinMain
```

6.4 给软中断挂钩

本节介绍给 Windows 软中断挂钩的方法。

这里给软中断处理挂上一个钩子，主要使用两个 VMM 提供的函数：Set_PM_Int_Vector 和 Hook_V86_Int_Chain。这两个函数一个是设置保护模式中断向量；另一个是挂住 V86 模式下的中断。

Set_PM_Int_Vector 设置保护模式下的中断，它的调用方法如下：

```
mov     eax, Interrupt      ;中断号
mov     cx, Segment         ;中断处理程序的段
mov     edx, Offset         ;中断处理程序的偏移
VMMCall Set_PM_Int_Vector
```

我们的中断处理程序不是要替代原来系统的中断处理程序，因此，在我们的处理程序中，当完成了处理任务后必须要跳回原来的处理程序去，这要使用 Simulate_Far_Jmp 函数：

```
mov     cx, Segment         ;远跳转的段
mov     edx, Offset         ;远跳转的偏移
VMMCall Simulate_Far_Jmp
```

由此可见，我们在调用 Set_PM_Int_Vector 设置中断向量之前必须先取得原中断向量，

这时可以用函数 Get_PM_Int_Vector：

```
mov      eax, Interrupt              ;中断号
VMMCall Get_PM_Int_Vector
mov      [Segment], cx               ;段地址
mov      [Offset], edx               ;偏移
```

对 V86 方式的中断挂钩则容易得多，因为 VMM 提供一个挂钩的函数 Hook_V86_Int_Chain：

```
mov    eax, Interrupt              ;中断号
mov    esi, OFFSET32 HookProc ;钩子过程的地址
VMMCall Hook_V86_Int_Chain
jc     not_installed
```

当动态 VxD 卸载时，就使用 Set_PM_Int_Vector 和 Unhook_V86_Int_Chain 来卸掉钩子。Unhook_V86_Int_Chain 的用法如下：

```
mov      eax, Interrupt                    ;中断号
mov      esi, OFFSET32 Hook_Proc           ;钩子过程的地址
VMMCall Unhook_V86_Int_Chain
jc       Error_Handler
```

下面的例子，挂住软中断 09H。

在 Windows 9X 中，中断 51H 才是键盘中断，但在一个虚拟机内运行的程序，却把中断 09H 当成是键盘的中断，这是 Windows 虚拟出来的假象。在 VxD 载入后，当用户按了 A 键时，就会弹出一个对话框，如图 6-6 所示。

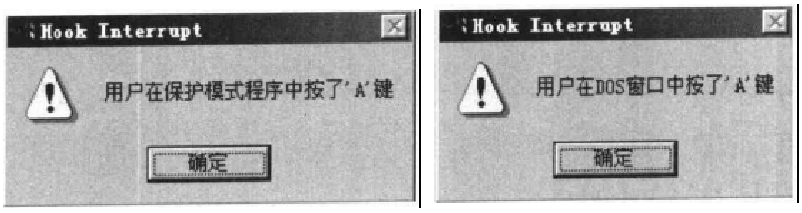


图 6-6

```
;**********************
;文件：HookInt.asm        *
;功能：对中断 09H 挂钩  *
;**********************
    .386p
    include C:\98DDK\inc\win98\vmm.inc
    include C:\98DDK\inc\win98\shell.inc
    include C:\98DDK\inc\win98\debug.inc

    INT_NUM = 09H              ;中断号
```

```
Declare_Virtual_Device          \
                    HOOKINT,    \
                    1, 0,       \
                    HOOKINT_control,      \
               Undefined_Device_ID, Undefined_Init_Order
;消息分派
BeginProc HOOKINT_control,LOCKED
     Control_Dispatch Sys_Dynamic_Device_Init, OnDynInitAsm, cCall
     Control_Dispatch Sys_Dynamic_Device_Exit, OnDynExitAsm, cCall
     Control_Dispatch W32_DeviceIOControl,     OnDynCtlAsm,   cCall
     clc
     ret
EndProc HOOKINT_control

;数据段
VxD_LOCKED_DATA_SEG
     Caption   db "Hook Interrupt",0
     VxDInit db "Beging Hook Interrupt.",0
     KeyPress_V86db "用户在 DOS 窗口中按了'A'键",0
     KeyPress_PM db "用户在保护模式程序中按了'A'键",0
align 4
     PreHookProc  dd ?

     Old_Sel   dW ?
     Old_Off   dd ?
VxD_LOCKED_DATA_ENDS

;代码段
VxD_LOCKED_CODE_SEG
BeginProc MsgBox              ;显示消息框
     VMMCall      Get_sys_vm_handle
     mov eax,MB_OK+MB_ICONEXCLAMATION
     mov edi,OFFSET32 Caption
     xor  esi,esi
     xor  edx,edx
     VxDCall  SHELL_Message
     ret
EndProc MsgBox
```

```
BeginProc HookProc          ;V86 的中断处理
    pushad
    pushf
    in   al,60H
    cmp al,1Eh              ;按了“A”键?
    jnz  HookProcOK
    mov ecx, OFFSET32 KeyPress_V86
    call  MsgBox
HookProcOK:
    popf
    popad
    stc
    ret
EndProc  HookProc

BeginProc PMProc            ;保护模式的中断处理
    pushad
    pushf
    in   al,60H
    cmp al,1Eh                  ;按了“A”键?
    jnz  PMProcOK
    mov ecx, OFFSET32 KeyPress_PM
    call  MsgBox
PMProcOK:
    popf
    popad
    mov cx,[Old_Sel]
    mov edx,[Old_Off]
    VMMjmp     Simulate_Far_Jmp       ;跳加原处理程序
EndProc PMProc

BeginProc OnDynInitAsm
    mov ecx, OFFSET32 VxDInit
    call  MsgBox

    mov eax,INT_NUM
    VMMCall     Get_PM_Int_Vector      ;取保护模式中断向量
```

```
        mov [Old_Sel],cx
        mov [Old_Off],edx
        mov esi,offset32 PMProc               ;分配回调函数
        VMMCall Allocate_PM_Call_Back
        mov       ecx,eax
        shr   ecx,16
        movzx     edx,ax
        mov       eax,INT_NUM
        VMMCall      Set_PM_Int_Vector        ;设置保护模式中断向量

        mov       eax, INT_NUM
        mov       esi, OFFSET32 HookProc
        VMMCall Hook_V86_Int_Chain            ;设置 V86 中断钩子

        mov eax,1
        ret
EndProc OnDynInitAsm

BeginProc OnDynExitAsm
        mov eax,INT_NUM
        mov cx,[Old_Sel]
        mov edx,[Old_Off]
        VMMCall      Set_PM_Int_Vector        ;设回原值

        mov       eax, INT_NUM
        mov       esi, OFFSET32 HookProc
        VMMCall Unhook_V86_Int_Chain          ;卸载 V86 中断钩子

        mov eax,1
        ret
EndProc OnDynExitAsm

BeginProc OnDynCtlAsm
        xor   eax,eax
        ret
EndProc OnDynCtlAsm

VxD_LOCKED_CODE_ENDS
```

end

要测试这个 VxD，需要在装入 VxD 后，用户按键盘，不能让 VxD 在装入后马上退出，因此前面的 VxD 测试程序不适用，在这里另写了一个测试程序。测试程序的外观如图 6-7 所示。

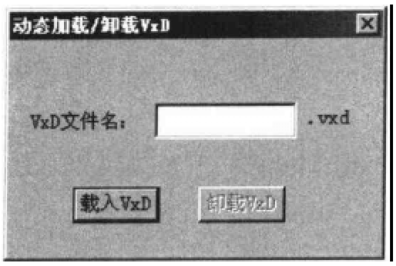


图 6-7

```
/*************************\
 *文件：LoadVxD.cpp       *
 *功能：动态载入/卸载 VxD *
\*************************/
#include "stdafx.h"
#include "resource.h"
BOOL CALLBACK MainDlgProc(HWND,UINT,UINT,LONG);

int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPreInst
LPSTR      lpCmdLine,   int   nCmdShow)
{
    DialogBox(hInstance,"MainDialog",0,MainDlgProc);
    return 0;
}

BOOL CALLBACK MainDlgProc(HWND hMainDlg,UINT uMsg,
                          UINT wParam,LONG lParam)
{
    static HANDLE hVxD = 0;

    switch(uMsg)
    {
    case WM_CLOSE:       //关闭主窗口
        if ( hVxD )
            SendMessage(hMainDlg,WM_COMMAND,IDC_UNLOAD,0);
        EndDialog(hMainDlg,TRUE);
```

```
            break;
        case WM_COMMAND:
            switch (wParam)
            {
            case IDC_LOAD:
                if ( hVxD )         break;
                char file[1024];
                char buff[512];
                GetDlgItemText(hMainDlg,IDC_VXDPATH,buff,256);
                wsprintf(file,"\\\\.\\%s.vxd",buff);
                //载入 VxD
                hVxD = CreateFile(file,0,0,0,0,FILE_FLAG_DELETE_ON_CLOSE,0);
                if ( hVxD == INVALID_HANDLE_VALUE )    //成功否?
                {
                    MessageBox(hMainDlg,"Load VxD Error","error",MB_OK);
                    ExitProcess(1);
                }
                EnableWindow(GetDlgItem(hMainDlg,IDC_LOAD),   FALSE);
                EnableWindow(GetDlgItem(hMainDlg,IDC_UNLOAD),TRUE );
                break;
                case IDC_UNLOAD:
                if ( !hVxD )   break;
                //卸载 VxD
                CloseHandle(hVxD);
                hVxD = NULL;
                EnableWindow(GetDlgItem(hMainDlg,IDC_LOAD),   TRUE );
                EnableWindow(GetDlgItem(hMainDlg,IDC_UNLOAD),FALSE);
                break;
            }
            break;
    }
    return FALSE;
}
```

资源文件不再列出。

6.5 虚拟化 IRQ

在前面章节中所介绍的方法一般不适用于硬件中断，要处理硬件中断，一般应该使

用 VPICD 提供的功能。VPICD 是 Windows 9X 本身提供的 VxD，提供虚拟的可编程中断控制器（Programmable Interrupt Controller）的操作服务，而让其他 VxD 请求、模拟硬件中断。

VPICD 模拟一个可编程中断控制器的功能。当 VPICD 初始化时，它已为每一个 IRQ 设置了默认的中断处理过程，这些中断处理过程把中断交给相应的虚拟机去处理，并要处理冲突。但其他的 VxD 可以“虚拟化”一个中断，当一个 IRQ 号被虚拟化后，VPICD 不再对它的中断调用默认的中断处理过程，这时就要由这个 VxD 去判断哪个虚拟机取得中断，并处理冲突。

虚拟化 IRQ 使用 VPICD 的服务 VPICD_Virtualize_IRQ，它的使用如下：

```
mov   edi, OFFSET32 vid
VxDcall VPICD_Virtualize_IRQ
jc        errorhandler
mov        [IRQHand], eax
```

返回的 eax 中放的是虚拟 IRQ 的句柄。参数 edi 中放的是一个 VPICD_IRQ_Descriptor 结构的地址：

```
VPICD_IRQ_Descriptor   STRUC
VID_IRQ_Number        dw  ?        ;中断号
VID_Options           dw  0
VID_Hw_Int_Proc       dd  ?        ;IRQ 硬件中断处理过程
VID_Virt_Int_Proc     dd  0        ;虚拟中断处理过程
VID_EOI_Proc          dd  0        ;EOI 命令处理过程
VID_Mask_Change_Proc      dd  0    ;中断屏蔽掩码改变的处理过程
VID_IRET_Proc         dd  0        ;虚拟机中断返回时调用
VID_IRET_Time_Out     dd  500      ;最大的中断超时时间（ms）
VID_Hw_Int_Ref        dd  ?        ;送给硬件中断处理程序的参数
VPICD_IRQ_Descriptor   ENDS
```

其中的 VID_Options 参数可以是以下的值的组合：

VPICD_OPT_READ_HW_IRR：读中断请求寄存器。

VPICD_OPT_CAN_SHARE：IRQ 可以共享，最多只能有 32 个 VxD 共享一个 IRQ。

VPICD_OPT_REF_DATA：使用 VID_Hw_Int_Ref 参数。

VPICD_OPT_VIRT_INT_REJECT：如果不想再虚拟化中断，则在中断处理过程中设置此位并设置 C 标志位（用 stc 指令）。

VPICD_OPT_SHARE_PMODE_ONLY：这个标志只在调试时使用。

下面的 VxD 程序虚拟化 IRQ 12。

在本书作者测试的机器上，IRQ 12 是 PS/2 兼容鼠标的中断。读者测试时可视自己机器上的中断分配情况来改变。

这个程序要用一个测试程序来测试。测试程序开始时装入 VxD，并用 Win32 API 函数 DeviceIOControl 把测试程序的主窗口句柄传给 VxD，当 VxD 处理硬件中断时，向测

试程序的主窗口发送一条用户自定义的消息。测试程序接到消息后在窗口上显示当前共接到了几次中断求。

下面是 VxD 源程序和测试程序，图 6-8 是测试程序的主窗口。

图 6-8

1. VxD 源程序

```
;********************
;文件：VirtIRQ.asm     *
;功能：虚拟化 IRQ      *
;********************
        .386p
        include C:\98DDK\inc\win98\vmm.inc
        include C:\98DDK\inc\win98\vpicd.inc
        include C:\98DDK\inc\win98\shell.inc
        include C:\98DDK\inc\win98\debug.inc

        WM_USER             = 400h
        WM_MY_MSG           = WM_USER + 1
        CONTROL_CODE        = 12345
        IRQ_NUM             = 12

DIOCPARAMETERS struc
        Internal1       dd ?
        VMHandle        dd ?
        Internal2       dd ?
        dwIoControlCode     dd ?
        lpvInBuffer     dd ?
        cbInBuffer      dd ?
        lpvOutBuffer    dd ?
        cbOutBuffer     dd ?
        lpcbBytesReturned   dd ?
        lpoOverlapped       dd ?
        hDevice             dd ?
```

```
    tagProcess          dd ?
  DIOCPARAMETERS ends
  Declare_Virtual_Device \
                    VIRTIRQ, \
                    1, 0, \
                    VIRTIRQ_control, \
               Undefined_Device_ID, Undefined_Init_Order
  ;消息分派
  BeginProc VIRTIRQ_control,LOCKED
      Control_Dispatch Sys_Dynamic_Device_Init, OnDynInitAsm, cCall
      Control_Dispatch Sys_Dynamic_Device_Exit, OnDynExitAsm, cCall
      Control_Dispatch W32_DeviceIOControl,      OnDynCtlAsm,   cCall
      clc
      ret
  EndProc VIRTIRQ_control

  ;数据段
  VxD_PAGEABLE_DATA_SEG
  IRQHand dd ?
  vid     VPICD_IRQ_Descriptor<IRQ_NUM,     VPICD_OPT_CAN_SHARE,OFFSET32
HWProc,0,0,0,0,500,0>
      hwnd     dd -1
      szCaptiondb "Virtual IRQ",0
      szInitOK  db "Init OK",0
      szInitFail db "Init Fail",0
  VxD_PAGEABLE_DATA_ENDS
  BeginProc MsgBox,LOCKED
      VMMCall     Get_sys_vm_handle          ;显示消息框
      mov eax,MB_OK+MB_ICONEXCLAMATION
      mov edi,OFFSET32 szCaption
      xor  esi,esi
      xor  edx,edx
      VxDCall  SHELL_Message
      ret
  EndProc MsgBox

  BeginPRoc HWProc,LOCKED                  ;硬件中断处理
      cmp hwnd,-1
```

```
        jz    HWProcExit
        pushad
        VxDcall _SHELL_PostMessage <hwnd,SPM_UM_DoNotWaitForCrit or WM_MY
_MSG,0,0,0,0>
        popad
    HWProcExit:
        stc
        ret
    EndProc  HWProc

    BeginPRoc VTProc,LOCKED
        ret
    EndProc  VTProc

    VxD_PAGEABLE_CODE_SEG
    BeginProc OnDynInitAsm
        mov     edi, OFFSET32 vid
        VxDcall VPICD_Virtualize_IRQ            ;虚拟化 IRQ
        jc      InitFail
        mov     [IRQHand], eax
        lea  ecx,szInitOK
        jmp  CallMsgBox
    InitFail:
        lea  ecx,szInitFail
    CallMsgBox:
        call  MsgBox
        mov eax,1
        ret
    EndProc OnDynInitAsm

    BeginProc OnDynExitAsm
        cmp IRQHand,0
        jz    Exit
        mov eax, [IRQHand]
        VxDcall VPICD_Force_Default_Behavior
    Exit:
        mov eax,1
        ret
```

```
EndProc OnDynExitAsm
BeginProc OnDynCtlAsm
      cmp [esi].dwIoControlCode,CONTROL_CODE
      jnz   CtlExit                ;取主窗口句柄
      mov eax,[esi].lpvInBuffer
      mov hwnd,eax
CtlExit:
      xor   eax,eax
      ret
EndProc OnDynCtlAsm
VxD_PAGEABLE_CODE_ENDS
      end
```

2. 测试程序

```
/**************************\
 *文件：TestVIRQ.cpp        *
 *功能：虚拟 IRQ 的测试程序*
\**************************/
#include "stdafx.h"
#include "resource.h"
#define CONTROL_CODE 12345
#define WM_MY_MSG WM_USER + 1
#define IRQ_NUM      12
BOOL CALLBACK MainDlgProc(HWND,UINT,UINT,LONG);
int APIENTRY WinMain(HMODULE hModule,HINSTANCE hNull,
                     PSTR pszCmdLine,int nCmdShow)
{
     DialogBox(hModule,"MainDialog",0,MainDlgProc);
     return 0;
}

BOOL CALLBACK MainDlgProc(HWND hMainDlg,UINT uMsg,
                          UINT wParam,LONG lParam)
{
     static unsigned long times = 0;
     static HANDLE hvxd = 0;
     char buffer[256];

     switch(uMsg)
```

```
{
case WM_INITDIALOG:
    wsprintf(buffer,"测试虚拟 IRQ %d",IRQ_NUM);
    SetWindowText(hMainDlg,buffer);
    break;
case WM_CLOSE:          // 主窗口关闭
    SendMessage(hMainDlg,WM_COMMAND,IDC_UNLOAD,0);
    EndDialog(hMainDlg,TRUE);
    break;
case WM_MY_MSG:         //VxD 发来消息，有中断发生
    times ++;
    wsprintf(buffer,"监测到中断 %lu 次",times);
    SetDlgItemText(hMainDlg,IDC_TIMES,buffer);
    break;
case WM_COMMAND:
    switch (wParam)
    {
    case IDC_LOAD:
        if ( hvxd )     break;
        //载入 VxD
        hvxd = CreateFile("\\\\.\\VirtIRQ.vxd", 0, 0, NULL, 0,
                    FILE_FLAG_DELETE_ON_CLOSE, NULL);
        if ( hvxd == INVALID_HANDLE_VALUE )
        {
            MessageBox(hMainDlg,"Load VxD Error","error",MB_OK);
            ExitProcess(1);
        }
        BOOL b;
        b = DeviceIoControl(hvxd,CONTROL_CODE,
                    (LPVOID)hMainDlg,0,0,0,0,0);
        if ( !b )
        {
            MessageBox(hMainDlg,"IO CONTROL","error",MB_OK);
            ExitProcess(1);
        }
        EnableWindow(GetDlgItem(hMainDlg,IDC_LOAD),   FALSE);
        EnableWindow(GetDlgItem(hMainDlg,IDC_UNLOAD),TRUE );
        break;
```

```
            case IDC_UNLOAD:
                if ( !hvxd )    break;
                CloseHandle(hvxd);          //卸载 VxD
                hvxd = NULL;
                EnableWindow(GetDlgItem(hMainDlg,IDC_LOAD),  TRUE );
                EnableWindow(GetDlgItem(hMainDlg,IDC_UNLOAD),FALSE);
                break;
            }
            break;
        }
        return FALSE;
}
```

需注意的是，这里的 IRQ 原来必须是共享的，若它被原来的 VxD 设置为不能共享（即 VID_Options 项没有 VPICD_OPT_CAN_SHARE 标志），则初始化会失败。若用户想知道自己的机器上哪些 IRQ 是共享的，可以在 SofeIce 或 TR for windows 中用“.vpicd”命令的第 3 项功能来查看。这是 VPICD 提供的查看接口，列出了系统中所有的 IRQ 处理过程的信息。如下是本书作者的计算机中的情况：

```
:.vpicd

***VPICD DEBUG INFORMATON ***
 [1] Global PIC information
[2] Per-VM PIC information
[3] IRQ handler information

>3
IRQ 00: Hw_Int_Proc=VTD_Get_Real_Time+0228, Refdata=C015A2C8, non-share
        Virt_Int_Proc=VTD_Get_Real_Time+02D2,
        EOI_Proc=VTD_Get_Real_Time+0310
        Mask_Change_Proc=#0028:00000000, IRET_Proc=VTD_Get_Real_Time+0316

IRQ 01: Hw_Int_Proc=#0028:C018A178, Refdata=00000000, non-share
        Virt_Int_Proc=#0028:C0189B1C, EOI_Proc=#0028:C0189AEC
        Mask_Cbange_Proc=#0028:00000000, IRET_Proc=#0028:C0189B38

IRQ 02: Hw_Int_Proc=#0028:C0157F6D, Refdata=00000000, non-share
        Virt_Int_Proc=#0028:00000000, EOI_Proc=Set_PM_Int_Vector+008A
        Mask_Change_Proc=Hook_V86_Int_Chain+01EF,
        IRET_Proc=#0028:C0157F6D
```

IRQ 03: Hw_Int_Proc=Set_PM_Int_Vector+00AE, Refdata=00000000, SHARE
Virt_Int_Proc=#0028:00000000, EOI_Proc=Set_PM_Int_Vector+00FA
Mask_Change_Proc=Set_PM_Int_Vector+011A,
IRET_Proc=Set_PM_Int_Vector+022

IRQ 04: Hw_Int_Proc=Set_PM_Int_Vector+00AE, Refdata=00000000, SHARE
Virt_Int_Proc=#0028:00000000, EOI_Proc=Set_PM_Int_Vector+00FA
Mask_Change_Proc=Set_PM_Int_Vector+011A,
RET_Proc=Set_PM_Int_Vector+022

IRQ 05: Hw_Int_Proc=Set_PM_Int_Vector+00AE, Refdata=00000000, non-share
Virt_Int_Proc=#0028:00000000, EOI_Proc=Set_PM_Int_Vector+00FA
Mask_Change_Proc=Set_PM_Int_Vector+011A,
IRET_Proc=Set_PM_Int_Vector+022

IRQ 06: Hw_Int_Proc=#0028:D23D6AD0, Refdata=00000000, SHARE
Virt_Int_Proc=#0028:00000000, EOI_Proc=#0028:D23D6ADD
Mask_Change_Proc=#0028:00000000, IRET_Proc=#0028:00000000

IRQ 06: Hw_Int_Proc=Set_PM_Int_Vector+00AE, Refdata=00000000, SHARE
Virt_Int_Proc=#0028:00000000, EOI_Proc=Set_PM_Int_Vector+0106
Mask_Change_Proc=#0028:00000000, IRET_Proc=Set_PM_Int_Vector+022C

IRQ 07: Hw_Int_Proc=#0028:D2437DC4, Refdata=3F9DF401, SHARE
Virt_Int_Proc=#0028:00000000, EOI_Proc=#0028:00000000
Mask_Change_Proc=#0028:00000000, IRET_Proc=#0028:00000000

IRQ 07: Hw_Int_Proc=Set_PM_Int_Vector+02AF, Refdata=00000000, SHARE
Virt_Int_Proc=#0028:00000000, EOI_Proc=Set_PM_Int_Vector+0260
Mask_Change_Proc=#0028:00000000, IRET_Proc=Set_PM_Int_Vector+0294

IRQ 08: Hw_Int_Proc=Set_PM_Int_Vector+00AE, Refdata=00000000, non-share
Virt_Int_Proc=#0028:00000000, EOI_Proc=Set_PM_Int_Vector+00FA
Mask_Change_Proc=Set_PM_Int_Vector+011A,
IRET_Proc=Set_PM_Int_Vector+022

IRQ 09: Hw_Int_Proc=_CONFIGMG_Set_Bus_Info+005C, Refdata=00000000, SHARE, SORTABLE

Virt_Int_Proc=#0028:00000000, EOI_Proc=#0028:00000000
Mask_Change_Proc=#0028:00000000, IRET_Proc=#0028:00000000

IRQ 09: Hw_Int_Proc=Set_PM_Int_Vector+00AE, Refdata=00000000, SHARE
Virt_Int_Proc=#0028:00000000, EOI_Proc=Set_PM_Int_Vector+0106
Mask_Change_Proc=#0028:00000000, IRET_Proc=Set_PM_Int_Vector+022C

IRQ 0A: Hw_Int_Proc=#0028:D2437854, Refdata=00000000, SHARE
Virt_Int_Proc=#0028:00000000, EOI_Proc=#0028:D2437DA0
Mask_Change_Proc=#0028:D2437DC0, IRET_Proc=#0028:00000000

IRQ 0A: Hw_Int_Proc=#0028:C012AC22, Refdata=C16BE610, SHARE, SORTABLE
Virt_Int_Proc=#0028:00000000, EOI_Proc=#0028:00000000
Mask_Change_Proc=#0028:00000000, IRET_Proc=#0028:00000000

IRQ 0A: Hw_Int_Proc=Set_PM_Int_Vector+02AF, Refdata=00000000, SHARE
Virt_Int_Proc=#0028:00000000, EOI_Proc=Set_PM_Int_Vector+0260
Mask_Change_Proc=#0028:00000000, IRET_Proc=Set_PM_Int_Vector+0294

IRQ 0B: Hw_Int_Proc=#0028:C170629B, Refdata=444F434C, SHARE
Virt_Int_Proc=#0028:C1706AE5, EOI_Proc=#0028:C1706A7A
Mask_Change_Proc=#0028:C1706AE5, IRET_Proc=#0028:C1706AE5

IRQ 0B: Hw_Int_Proc=Set_PM_Int_Vector+02AF, Refdata=00000000, SHARE
Virt_Int_Proc=#0028:00000000, EOI_Proc=Set_PM_Int_Vector+0260
Mask_Change_Proc=#0028:00000000, IRET_Proc=Set_PM_Int_Vector+0294

IRQ 0C: Hw_Int_Proc=VMD_Post_Pointer_Message+0254, Refdata=00000000, SHARE
Virt_Int_Proc=#0028:00000000, EOI_Proc=#0028:00000000
Mask_Change_Proc=VPICD_VM_SlavePIC_Mask_Change+0014,
IRET_Proc=#0028:000

IRQ 0C: Hw_Int_Proc=Set_PM_Int_Vector+00AE, Refdata=00000000, SHARE
Virt_Int_Proc=#0028:00000000, EOI_Proc=Set_PM_Int_Vector+0106
Mask_Change_Proc=#0028:00000000, IRET_Proc=Set_PM_Int_Vector+022C

IRQ 0D: Hw_Int_Proc=_VMCPD_Get_FP_Instruction_Size+0250, Refdata=00000000, SHARE, SHARE-PM

```
            Virt_Int_Proc=_VMCPD_Get_FP_Instruction_Size+02D0,
    EOI_Proc=_VMCPD_Get_F
            Mask_Change_Proc=#0028:00000000, IRET_Proc=#0028:00000000
    IRQ 0D: Hw_Int_Proc=Set_PM_Int_Vector+0111,Refdata=00000000,SHARE,SHARE-PM
            Virt_Int_Proc=#0028:00000000, EOI_Proc=Set_PM_Int_Vector+0106
            Mask_Change_Proc=#0028:00000000, IRET_Proc=Set_PM_Int_Vector+022C
    IRQ 0E: Hw_Int_Proc=#0028:D23D8498, Refdata=D23D3000, SHARE
            Virt_Int_Proc=#0028:00000000, EOI_Proc=#0028:D23D862F
            Mask_Change_Proc=#0028:00000000, IRET_Proc=#0028:00000000
    IRQ 0E: Hw_Int_Proc=Set_PM_Int_Vector+00AE, Refdata=00000000, SHARE
            Virt_Int_Proc=#0028:00000000, EOI_Proc=Set_PM_Int_Vector+0106
            Mask_Change_Proc=#0028:00000000, IRET_Proc=Set_PM_Int_Vector+022C
    IRQ 0F: Hw_Int_Proc=#0028:D23D8498, Refdata=D23D48C0, SHARE
            Virt_Int_Proc=#0028:00000000, EOI_Proc=#0028:D23D862F
            Mask_Change_Proc=#0028:00000000, IRET_Proc=#0028:00000000
    IRQ 0F: Hw_Int_Proc=Set_PM_Int_Vector+00AE, Refdata=00000000, SHARE
            Virt_Int_Proc=#0028:00000000, EOI_Proc=Set_PM_Int_Vector+0106
            Mask_Change_Proc=#0028:00000000, IRET_Proc=Set_PM_Int_Vector+022C
```

6.6 I/O 操作

6.6.1 I/O 操作概述

I/O 操作是系统对外部设备进行控制的手段。每个设备都占用系统的一个或几个端口，设备驱动程序通过对相应的端口进读/写操作来获得设备的状态和控制设备的动作。如图 6-9 所示的是在 Windows 98 中列出的系统各端口所对应的设备情况。

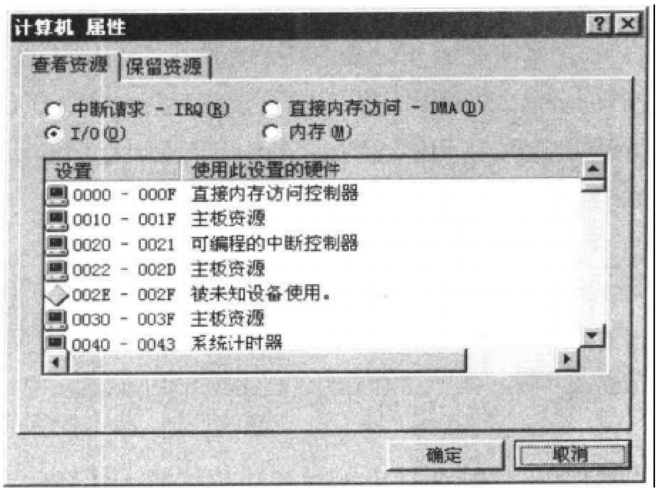


图 6-9

一个健壮的操作系统应该限制用户态程序的 I/O 操作。这是因为如果任凭用户态程序作 I/O 操作，就相当于放任用户态程序绕过操作系统而直接控制外部设备，系统的安全性就完全没有保障了。而且如果多个用户态程序同时对一个外部设备进行操作，设备的动作亦会一片混乱，虚拟机的概念也不复存在了。

在 Windows 中，操作系统可以截获对端口的操作，设备驱动程序可以为对应的端口操作安装处理函数。通过为设备的端口安装处理函数，设备驱动程序就可以模拟一个虚的设备，解决各虚拟机间资源占用的冲突，使得各虚拟机都好像独占该设备而不是与别的虚拟机分享。

在 80X86 系统中，程序通过 IN、OUT、INS、OUTS 等 I/O 指令对端口进行读/写操作。在保护模式中，CPU 执行这些指令时，首先会检查标志寄存器 EFLAG 的 IOPL（I/O Privilege Level）位。这个位标志着第几环的代码可以执行 I/O 指令，如果当前特权级的值小于或等于这个值时，I/O 指令可以正常执行，否则就要检查 IO 许可位图。

IO 位图是任务状态段的一部分，它是二进制位串，这个二进制位串的每一个二进制位顺序对应一个端口。在执行 IN、OUT 等指令而当前特权级的值大于 IOPL 时，CPU 查 IO 位图，当对应的二进制值为 0 时，则允许访问，否则访问将引起异常。例如，IO 位图的第 0 个字节的第 0 位为 0 时，表示端口 0 可以访问，如果第 0 个字节的第 7 位是 1，则表示访问端口 7 会引起异常。

6.6.2 Windows 9X 的 I/O 操作

在 Windows 9X 中，普通用户程序运行在特权级 3 中，而当在执行普通用户程序时，EFLAG 的 IOPL 值是 0。因此，用户程序能否访问 I/O 端口则取决于 IO 位图。VxD 通过调用 VMM 的函数 Install_IO_Handler 或 Install_Mult_IO_Handlers 来安装对指定端口的陷阱处理函数。这两个函数会设置 IO 位图，并设置相应端口的陷阱处理函数。注意，每个端口只能安装一个处理过程，即 VxD 之间不能共享端口。你可以在 SofeIce 或 Tr for win 中用“tss”命令来查看系统中端口陷阱处理的过程，下面是 tss 命令的一部分输出：

```
:tss
TR=0018   BASE=C000D7A4   LIMIT=2069
LDT=0000 GS=0000 FS=0000 DS=0000 SS=0000 CS=0000   ES=0000   CR3=00000000
EAX=00000000   EBX=00000000   ECX=00000000   EDX=00000000   EIP=00000000
ESI=00000000   EDI=00000000   EBP=00000000   ESP=00000000   EFL=00000000
SS0=0030:C0FADFA8   SS1=0000:00000000   SS2=0000:00000000
I/O Map Base=0068   I/O Map Size=2000
Port   Handler   Trapped    Owner
0000   C015CE4A   Yes       VDMAD(01)+182A
0001   C015CEDA   Yes       VDMAD(01)+18BA
0002   C015CE4A   Yes       VDMAD(01)+182A
0003   C015CEDA   Yes       VDMAD(01)+18BA
```

```
0004  C015CE4A  Yes    VDMAD(01)+182A
0005  C015CEDA  Yes    VDMAD(01)+18BA
0006  C015CE4A  Yes    VDMAD(01)+182A
0007  C015CEDA  Yes    VDMAD(01)+18BA
0008  C015CBA5  Yes    VDMAD(01)+1585
0009  C015CD4A  Yes    VDMAD(01)+172A
000A  C015CD8A  Yes    VDMAD(01)+176A
000B  C015CCCB  Yes    VDMAD(01)+16AB
000C  C015CDCA  Yes    VDMAD(01)+17AA
000D  C015CC18  Yes    VDMAD(01)+15F8
000E  C015CC9B  Yes    VDMAD(01)+167B
000F  C015CC6D  Yes    VDMAD(01)+164D
0020  C01582E0  Yes    VPICD_Virtual_EOI+0058
0021  C01583DC  Yes    VPICD_Virtual_EOI+0154
0040  C0256380  Yes    _GetInstanceInfo+01E8
0043  C0256314  Yes    _GetInstanceInfo+017C
0060  C0189F44  Yes    VKD(01)+0C00
0061  C015E1C8  No     VSD_SoundOn+0001
0064  C0189E30  Yes    VKD(01)+0AEC
0070  D2437A88  Yes    ES1371(01)+00050268
0081  C015CF7F  Yes    VDMAD(01)+195F
......                 ; 这里略去一些输出
```

Install_IO_Handler 的用法如下：

```
mov     esi, IOCallback     ;端口陷阱处理函数
mov     edx, Port           ;端口号
VMMCall Install_IO_Handler
jc      not_installed
```

下面的 VxD 对 800H 端口安装陷阱处理函数。利用前面章节的测试程序装入这个 VxD 后，如果有用户程序访问了端口 800H，则系统会调用我们安装的处理函数，弹出一个对话框，如图 6-10 所示。并且，如果是 IN 指令，则根据输入指令要求输入字节、字或双字，而用户程序将得到的相应结果分别是 78H、5678H 或 12345678H。

图 6-10

```
;*****************************
;文件：IOTrap.asm               *
;功能：演示设置和处理 IO 陷阱 *
;*****************************
      .386p
      include C:\98DDK\inc\win98\vmm.inc
      include C:\98DDK\inc\win98\shell.inc
      include C:\98DDK\inc\win98\debug.inc
      PORT_NUM = 800H                    ;端口号

Declare_Virtual_Device        \
                    IOTRAP, \
                    1, 0,       \
                    IOTRAP_control,   \
               Undefined_Device_ID, Undefined_Init_Order
;消息分派
BeginProc IOTRAP_control,  LOCKED
     Control_Dispatch Sys_Dynamic_Device_Init, OnDynInit, cCall
     Control_Dispatch Sys_Dynamic_Device_Exit, OnDynExit, cCall
     Control_Dispatch W32_DeviceIOControl,        OnDynCtl,  cCall
     clc
     ret
EndProc IOTRAP_control

;数据段
VxD_PAGEABLE_DATA_SEG
     szCaptiondb "IO Trapping",0
     szText           db "Ring 3  程序访问  800H  端口",0
     szSuccessdb "Init OK",0
     szFail           db "Init Fail",0
VxD_PAGEABLE_DATA_ENDS

;代码段
VxD_PAGEABLE_CODE_SEG
BeginProc MsgBox
     VMMCall      Get_sys_vm_handle
     mov eax,MB_OK+MB_ICONEXCLAMATION
     mov edi,OFFSET32 szCaption
```

```
        xor   esi,esi
        xor   edx,edx
        VxDCall  SHELL_Message
        ret
EndProc MsgBox

BeginProc IOTrapProc                    ;IO 陷阱处理过程
        lea   ecx,szText
        call  MsgBox
        mov eax,12345678H                      ;设置返回值
        ret
EndProc IOTrapProc

BeginProc OnDynInit
        lea          esi, IOTrapProc
        mov                edx, PORT_NUM       ;安装 IO 陷阱处理过程
        VMMCall Install_IO_Handler
        mov ecx,OFFSET32 szSuccess
        jnc   install_ok
        mov ecx,OFFSET32 szFail
install_ok:
        call  MsgBox
        mov eax,1
        ret
EndProc OnDynInit

BeginProc OnDynExit
        mov          edx, PORT_NUM             ;卸装 IO 陷阱处理过程
        VMMCall Remove_IO_Handler
        mov eax,1
        ret
EndProc OnDynExit

BeginProc OnDynCtl
        xor   eax,eax
        ret
EndProc OnDynCtl

VxD_PAGEABLE_CODE_ENDS
```

end

要测试这个 VxD，可以在装入后，打开 DOS 窗口，用 Debug.exe 的 i 指令来测试：

```
C:\>debug
-i 800
78
-
```

由此可见，取得的结果是 78H。若要查看取双字节时的情形，要写一个小程序：

```
C:\>debug
-a 100
12C6:0100 mov dx,800
12C6:0103 in    ax,dx
12C6:0104 int 3
12C6:0105
-g=100

AX=5678   BX=0000   CX=0000   DX=0800   SP=FFEE   BP=0000   SI=0000
DI=0000
DS=12C6  ES=12C6  SS=12C6  CS=12C6  IP=0104   NV UP EI PL NZ NA PO NC
12C6:0104 CC              INT    3
-
```

得到 AX 中的结果是 5678H。取双字时的情形，读者可自己写个小程序试试。

之所以能有这样的效果，是因为处在 CPU 0 环的系统可以截取第 3 环的程序的 I/O 操作，如果 I/O 操作是在第 0 环进行的，系统就无法截取了。第 0 环的代码可以截取第 3 环代码的 I/O 操作，这是虚拟机的重要基础。

Windows 下的普通程序都可运行在虚拟机下，因此，如果你的程序进行 I/O 操作，则只是改变了特定虚拟机的 I/O 状态。

例如，你在 DOS 窗口中对 8253 计时器进行更改后，那么在这个窗口下运行依赖于时钟的 DOS 程序时，就会看到时钟的速度改变了，但在其他窗口和系统的时钟并不会改变。

这是因为每个 DOS 窗口都有一部虚拟机器，机器中的 8253 是虚拟的，而不是真实物理上的 8253 计时器。

但如果你的程序在 0 环上运行情况就不同了，因为这时系统无法截取程序对 I/O 端口的更改。

下面的程序进入 0 环后更改 8253 的计数初值。

```
/**********************************\
 *文件：C8253.CPP              *
 *功能：演示直接写 IO           *
\**********************************/
```

```
#include "stdafx.h"
#include <windows.h>
WORD Count;      //8253 计数初值

int main(int argc, char* argv[])
{
    Count = 0x800;

    _asm jmp Begin

    _asm
    {

Change8253:
    mov      al,0x36
    out      0x43,al
    mov      ax,Count
    out      0x40,al
    mov      al,ah
    out      0x40,al
    retf
    }

    __asm
    {

ToRing0Code:
    push ebp
    mov   ebp,esp
    sub   esp,8
    call  GetLdtAddress          //取出 LDT 的地址，返回结果在 eax 中
    mov ecx,[eax]                //保存 LDT 第一个描述符
    mov [ebp-4],ecx
    mov ecx,[eax+4]
    mov [ebp-8],ecx
    mov edx,[ebp+8]                 //把调用门的内容写入 LDT
    mov [eax],dx                    //偏移量的低 16 位
    mov word ptr [eax+2],28h        //段选择子
```

```
    mov word ptr [eax+4],0ec00h //属性
    shr   edx,16                //偏移量的高 16 位
    mov [eax+6],dx
    push eax

//  CALL32 7,0        //调用 Ring0 子程序
    _emit 0x9a
    _emit 0
    _emit 0
    _emit 0
    _emit 0
    _emit 7
    _emit 0
    pop  ebx
    mov edx,[ebp-4]          //恢复 LDT 第一个描述符
    mov [ebx],edx
    mov edx,[ebp-8]
    mov [ebx+4],edx
    leave
    ret 4

GetLdtAddress:
    push ebx                 //先要取 GDT 的地址
    sgdt [esp-2]
    pop  ebx
    sldt  ax                 //取 LDT 内容
    and  eax,0fff8H          //屏蔽掉低 3 位、eax 的高 16 位清 0
    add  ebx,eax             //算出 LDT 描述符的位置
    mov eax,[ebx+2]          //从描述符中取出 LDT 的地址
    mov dl,[ebx+7]
    shl  edx,24
    and  eax,0ffffffh
    or   eax,edx
    ret
    }

_asm
    {
```

```
Begin:
    pushad
    push offset Change8253
    call ToRing0Code
    popad
    }
    return 0;
}
```

程序在 0 环把 8253 的计数初值设为 800H，当打开 Windows 的时钟，会发现时钟明显加快了，若用户进行双击操作，则可能经常没有反应，因为 Windows 认为用户两次单击之间的时间太长，不是双击。

而如果用户把这个计数初值设大，例如 3000H，则时钟变慢，而且当用户操作最大、最小化窗口时，可以明显看到其动画过程。

第七章　文件系统

本章主要介绍 Win32 中对文件的操作方法和一些文件系统的原理。

7.1　Windows 9X 的 FAT 和长文件名

Win32 使用的文件系统有几种:FAT16、FAT32、NTFS 等。这些文件系统反映了 Microsoft 对系统性能改进和适应新的硬件高速发展的过程。要查看文件系统的属性，可以通过右击“我的电脑”中的相应盘符，选“属性”看到，如图 7-1 所示。

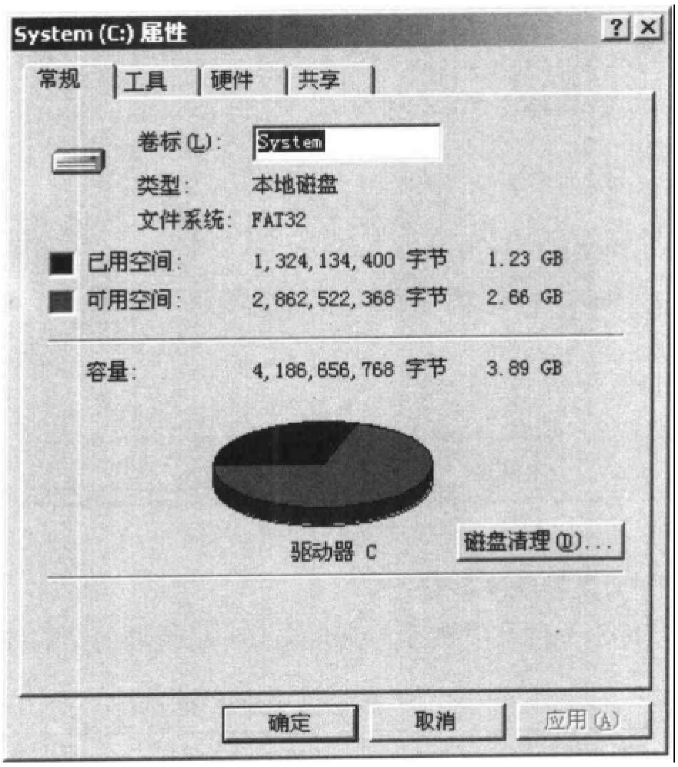


图 7-1

在 DOS/Windows 3.X 中，文件名的长度的限制在 8.3 格式，即前缀最长是 8，扩展名最长为 3。而 Windows 9X 的文件名长度可以是 255 字节。那么 Windows 如何支持长文件名？

为直观说明，我们首先要在 A 盘建立一个有长文件名的文件，起名为“This_File_Has_A_Long_FileName.txt”，列文件名，其内容如下：

C:\>dir a:

Volume in drive A has no label

Volume Serial Number is 1ED8-0C20

Directory of A:\

THIS_F~1 TXT 13 06-05-00 23:09
This_File_Has_A_Long_FileName.txt

1 file(s) 13 bytes
0 dir(s) 1,438,720 bytes free

或在 Windows 中查看如图 7-2 所示。

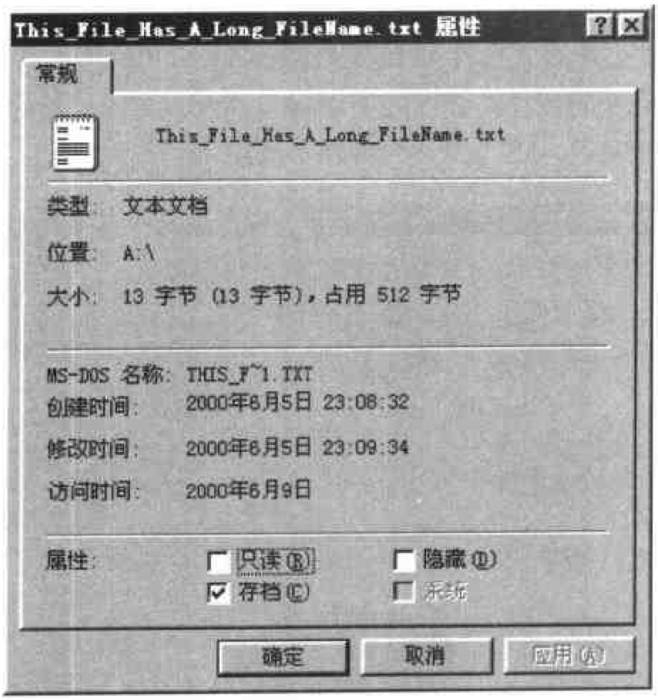


图 7-2

那么，Windows 如何支持长文件名？

如果您对旧版本 DOS 下目录项熟悉，就会知道，目录项的结构如图 7-3 所示。

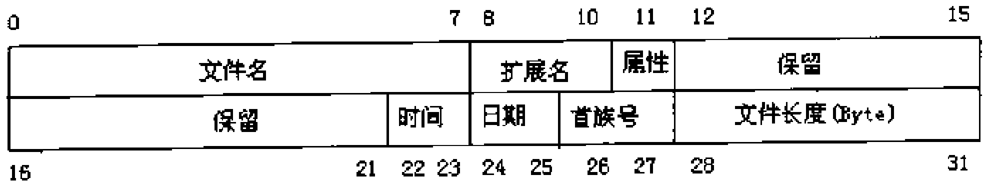


图 7-3

其中"属性"一项说明文件的属性如隐藏、系统、目录等，具体如图 7-4 所示。

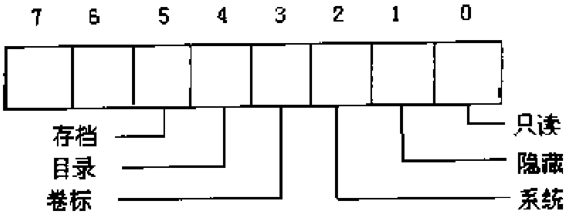


图 7-4

这些属性可以组合，例如存档、只读的文件属性是 21H。最常用属性的是 20H，即如我们上面所建的文件一样，只有存档属性。但并不是所有可能的组合都可用，例如，“卷标”位若置为 1，则只读、隐藏等位就没有意义了。

Windows 9X 正是利用这种特性来支持长文件名。它在生成一个文件的同时并不止生成一个目录项，而是又另外生成几个目录项来存放长文件名的信息，它们的属性是 0FH，即同时具有卷标、系统、隐藏、只读属性，这在以前的 DOS 或 Windows 版本中是不可能的，旧的 DOS 或 Windows 会忽略掉这些目录项。这些目录项我们称为 LFN（Long File Name）目录项。

那么，长文件名又是怎样存放在这些目录项中？现在我们来看看上面我们创建的文件的目录项：

```
A:\>debug
-l 400 0 13 1
-d 400
……
1273:0440   43 61 00 6D 00 65 00 2E-00 74 00 0F 00 F3 78 00      Ca.m.e...t....x.
1273:0450   74 00 00 00 FF FF FF FF-FF FF 00 00 FF FF FF FF      t...............
1273:0460   02 5F 00 41 00 5F 00 4C-00 6F 00 0F 00 F3 6E 00      ._.A._.L.o....n.
1273:0470   67 00 5F 00 46 00 69 00-6C 00 00 00 65 00 4E 00      g._.F.i.l...e.N.
-d
1273:0480   01 54 00 68 00 69 00 73-00 5F 00 0F 00 F3 46 00      .T.h.i.s._....F.
1273:0490   69 00 6C 00 65 00 5F 00-48 00 00 00 61 00 73 00      i.l.e._.H...a.s.
1273:04A0   54 48 49 53 5F 46 7E 31-54 58 54 20 00 40 10 B9      THIS_F~1TXT .@..
1273:04B0   C5 28 C5 28 00 00 31 B9-C5 28 02 00 0D 00 00 00      .(.(..1..(......
1273:04C0   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00      ................
……
```

我们可以很显然看到，长文件名是用 Unicode 表示的。LFN 目录项如图 7-5 所示。

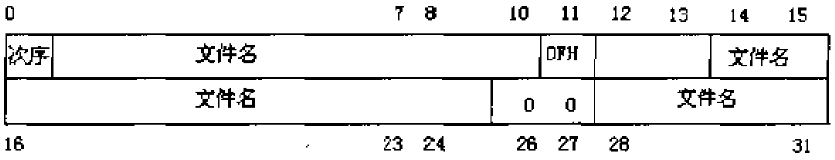


图 7-5

其中的第 0 个字节是本目录项的次序和结束标志。它的意义如图 7-6 所示。

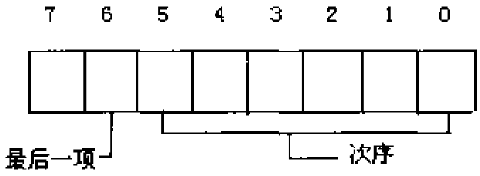


图 7-6

上面的例子中，次序分别是 1、2、3，最后一项的第 6 位是 1，因此变成 43H 了。

LFN 目录中文件名被分成三份，共可存放 13 个 Unicode 字符：

第一部分是 1~10 字节，共 10 字节（5 个 Unicode）。

第二部分是 14~25，12 字节（6 个 Unicode）。

第三部分是 28~31 字节，共 4 字节（2 个 Unicode）。

而第 11 字节恒为 0FH，26、27 字节恒为 0，12、13 字节对同一文件是一样的，估计是校验和。

7.2 文件操作

若用 C/C++语言写程序，你当然可以用标准的 C 文件操作库函数来操纵文件，但 Windows 提供了灵活的文件操作 API，使用它们可以写出更符合 Windows 标准和发挥 Windows 功能的程序来。下面介绍一些常用的文件 API。

7.2.1 建立文件

建立文件，可通过 CreateFile 调用：

call CreateFile, offset FileName, AccessFlg, ShareMode, offset Secret, CreateFlg, FileAttr, hFile

CreateFile 成功时返回文件的名柄，失败返回值是 INVALID_HANDLE_VALUE，这个值被定义为-1。

这个 API 的功能并不止是建立文件这么简单，它可以创建或打开文件、逻辑盘、物理盘、串/并口、套接字、管道等。这跟 Unix 中把设备看成文件是同一概念。我们在这里只讨论普通文件的操作。

现在解释它的参数：

1）FileName 是要打开的文件名。

2）AccessFlg 说明你要打开文件做什么。可以是：

① 0。

② GENERIC_READ，读文件。

③ GENERIC_WRITE，写文件。

④ ②和③的和，既读也写。

3）ShareMode 是你想文件的共享方式，可以是：

① 0，独占模式。

② FILE_SHARE_READ，只读。

③ FILE_SHARE_WRITE，只写。

④ ②和③的和，可读可写。

4）Secret 是指向文件的的安全属性的结构。

5）CreateFlg 可以是以下的值：

① CREATE_NEW 建立新文件，若文件存在，则失败。

② CREATE_ALWAYS 不管文件是否存在都建立新文件。

③ OPEN_EXIST 打开文件。

④ OPEN_ALWAYS，不管文件是否存在都有打开。若不存在则创建它。

⑤ TRUNCATE_EXISTING 此标志要与 CREATE_WRITE 同用，表示打开一个已存在的文件，并将其大小置 0。

6）FileAttr 文件属性标志。可以是存档、隐藏等。

7）hFile 指定文件句柄使建立的文件属性与它相同。

7.2.2　定位文件指针函数

很多时候我们未必会按顺序从头到尾地写文件，这时要用到时定位文件指针函数：

call SetFilePointer, hFile, Distance, lpDistanceHigh, Method

这个 API 成功则返回指针的原来位置，否则返回-1。

1）hFile 是文件的句柄。

2）Distance 是移动的距离，一般可正可负（即-2G~+2G）。

3）LpDistanceHigh 是当要移动的距离超过 2G 时，这个参数指向距离的高位 32 位，并且返回时被填入原指针的高 32 位。

4）Method 是指出 Distance 和 lpDistanceHigh 相对于什么位置而言，可以是以下的值之一：

① FILE_BEOIN，由文件头开始计算，这时 Distance 和 lpDistanceHight 指出的值是无符号的。

② FILE_CURRENT，由当前指针计算。

③ FILE_END，由文件末尾开始，这时距离应为负值。

7.2.3　写文件

写文件的模式如下：

call WriteFile,hFile,offset Buffer,nByte,offset Writed,offset Overlapped

这个 API 成功时返加非 0，否则返回 0。

1）hFile，文件句柄。

2）Buffer，要写的内容。

3）nByte，要写的字节数。

4）Writed，指向写了的字节数，由 WriteFile 填写。

5）LpOverlapped，这个参数与异步 IO 有关，可置为 NULL。

7.2.4　读文件

读文件的模式如下：

call ReadFile,hFile,offset Buffer,nByte,offset Readed,offset Overlapped

这个 API 成功时返加非 0，否则返回 0。

参数的意义与 WriteFile 基本相同。

另外还有 WrietFileEx 和 ReadFileEx，是专门为异步操作提供的，这时就不细说了。

7.2.5 关闭文件

关闭文件的模式如下：

call CloseHandle,hFile

这个 API 我们在以前的章节已多次使用，不再赘述。

7.2.6 例程

下面的例子建立一个文件名为 Testing.txt，并写入字串“文件操作测试”。

```
;*****************************
;FileOper.asm                *
;功能：演示文件操作           *
;*****************************

.386p
.model flat,stdcall          ;平坦内存模式

;常量定义

NULL   = 0
MB_OK = 0
INVALID_HANDLE_VALUE = -1
GENERIC_READ     = 80000000H
GENERIC_WRITE = 40000000H
FILE_SHARE_READ       = 1
FILE_SHARE_WRITE     = 2
FILE_SHARE_DELETE = 4
OPEN_EXISTINO = 3
CREATE_NEW    = 1

;外部过程声明
extrn MessageBoxA:proc
extrn ExitProcess:proc
extrn CreateFileA:proc
extrn ReadFile:proc
extrn WriteFile:proc
```

```
extrn CloseHandle:proc

.data
      Caption db 'OK',0
      Text db '建立文件、写文件成功',0
      ErrCaption     db 'ERROR',0
      ErrCreate db '建立文件错误',0
      ErrWrite  db '写文件错误',0
      FileContext          db '文件操作测试',0
      FILELENGTH = $ - FileContext
      FileNamedb 'Testing.txt'
      hFile      dd ?
      align 4
      write      dd ?
.code
main:
      ;建立文件
      call   CreateFileA,offset   FileName,GENERIC_WRITE,FILE_SHARE_READ   OR
FILE_SHARE_WRITE,NULL,CREATE_NEW,NULL,NULL
      mov   [hFile],eax
      cmp   eax,INVALID_HANDLE_VALUE
      jnz   Write
      call MessageBoxA,NULL,offset ErrCreate,offset ErrCaption,MB_OK
      jmp Exit
Write:
      ;写文件
      call WriteFile,eax,offset FileContext,FILELENGTH,offset write,NULL
      cmp eax,0
      jnz OK
      call CloseHandle ,[hFile]
      call MessageBoxA,NULL,offset ErrWrite,offset ErrCaption,MB_OK
      jmp Exit
OK:
      ;显示成功信息
      call MessageBoxA,NULL,offset Text,offset Caption,MB_OK
      ;关闭文件句柄
      call CloseHandle ,[hFile]
Exit:
```

```
call ExitProcess,0
end main
```

7.3 Windows NT 下绝对磁盘的读写

前面我们提到 CreateFile 可以打开逻辑盘或物理盘，现在我们看如何利用这点来直接读写磁盘。

在 Windows NT（Windows 95/98 不支持直接用 CreateFile 打开磁盘）中，在 CreatFile 时把路径名设为“\\.\x:”时，则打开一个逻辑盘，其中 x 是 C、D、E 等盘符。

下面的程序在 NT 下运行，读取 C 盘的引导记录（1 面 0 道 1 扇区）。

```
;*****************************
;AbsDiskNT.asm               *
;功能：Windows NT 中绝对      *
;          磁盘读写            *
;*****************************

.386p
.model flat,stdcall          ;平坦内存模式

;常量定义
NULL    =0
MB_OK = 0
INVALID_HANDLE_VALUE = -1
GENERIC_READ   = 80000000H
GENERIC_WRITE = 40000000H
FILE_SHARE_READ    = 1
FILE_SHARE_WRITE   = 2
FILE_SHARE_DELETE = 4
OPEN_EXISTING = 3

;外部过程声明
extrn MessageBoxA:proc
extrn ExitProcess:proc
extrn CreateFileA:proc
extrn ReadFile:proc
extrn CloseHandle:proc

.data
```

```
        FileName db '\\.\C:',0
        align 4                    ;双字对齐
        readed     dd 0
        Buffer   db 4096 dup (0)
        hFile      dd 0

        Caption   db 'NT 中绝对磁盘读写',0
        ErrCreate db '建文件错!',0
        ErrRead  db '读盘错误！ ',0

        ShowTextdb 4096*3 dup (0)
        Digit      db '0123456789ABCDEF'

    .code
    main:
        ;建立文件
        call   CreateFileA,offset   FileName,GENERIC_READ,   FILE_SHARE_READ   OR
FILE_SHARE_WRITE,NULL,OPEN_EXISTING,NULL,NULL
        mov    [hFile],eax
        cmp    eax,INVALID_HANDLE_VALUE
        jnz    read
        ;显示错误信息
        call ShowError,offset ErrCreate
    read:
        ;读数据
        call ReadFile,eax,offset Buffer,512,offset readed,NULL
        cmp eax,0
        jnz    show
        ;显示错误信息
        call ShowError,offset ErrRead

    show:
        ;显示读出的内容
        call ShowBuffer
        ;关闭文件句柄
        call CloseHandle ,[hFile]
        ;退出
        call ExitProcess,0
```

```
ShowError proc ,MESSAGE:DWORD  ;显示出错信息并退出
      call MessageBoxA,NULL,MESSAGE,offset Caption,MB_OK
      cmp [hFile],0
      jz    endShowErr
      call CloseHandle ,[hFile]      ;关闭句柄
endShowErr:
      call ExitProcess,0             ;退出
ShowError endp
ShowBuffer proc                ;显示所读出的信息

;把数据转换成 16 进制的形式
      mov esi,offset Buffer          ;数据
      mov edi,offset ShowText        ;转换后的数据
      mov ebx,offset Digit
      mov ecx,0
      xor eax,eax
computeAgain:
      cmp [readed],0
      jz endCompute
      dec [readed]
      mov al,[esi]
      push eax
      shr eax,4                      ;高 4 位
      mov al,[ebx+eax]
      mov byte ptr[edi],al
      inc edi
      pop eax
      and eax,0fH                    ;低 4 位
      mov al,[ebx+eax]
      mov byte ptr[edi],al
      inc edi
      mov byte ptr[edi],' '          ;空格
      inc edi
      inc esi
      inc ecx
      cmp ecx,16
      jnz computeAgain
      xor ecx,ecx
```

```
        mov byte ptr[edi-1],13          ;回车
        jmp computeAgain

endCompute:
        ;显示
        call MessageBoxA,NULL,offset ShowText,offset Caption,MB_OK
        ret
ShowBuffer endp
end main
```

程序用一个 MessageBox 显示了 C 盘引导纪录的内容，如图 7-7 所示。

图 7-7

我们可以用 Debug 调出引导记录来验证一下我们的程序是否正确。但要注意：Windows NT 不支持用 int 13 直接读硬盘，下面的 Debug 程序必须在 Windows 9X 的 DOS 窗口或纯 DOS 下运行，否则会出现如图 7-8 所示的警告。

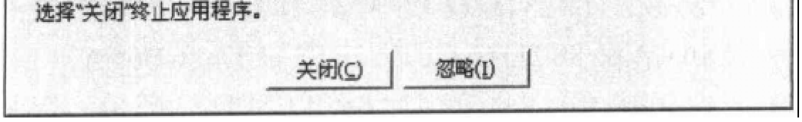


图 7-8

```
C:\>debug
-a
1273:0100 mov ax,201
1273:0103 mov bx,200
1273:0106 mov cx,1
1273:0109 mov dx,180
1273:010C int 13
1273:010E int 20
1273:0110
-g=100

Program terminated normally
-d 200
1273:0200   EB 58 90 4D 53 57 49 4E-34 2E 31 00 02 08 20 00   .X.MSWIN4.1.....
1273:0210   02 00 00 00 00 F8 00 00-3F 00 FF 00 3F 00 00 00   ........?...?...
1273:0220   3F 04 7D 00 3A 1F 00 00-00 00 00 00 02 00 00 00   ?.}.:...........
1273:0230   01 00 06 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1273:0240   80 00 29 E7 16 6D 25 28-43 29 20 20 20 20 20 20   ..)..m%(C)
1273:0250   20 20 46 41 54 33 32 20-20 20 33 C9 8E D1 BC F4   FAT32    3.....
1273:0260   7B 8E C1 8E D9 BD 00 7C-88 4E 02 8A 56 40 B4 08   {......|.N..V@..
1273:0270   CD 13 73 05 B9 FF FF 8A-F1 66 0F B6 C6 40 66 0F   ..s......f...@f.
-d
1273:0280   B6 D1 80 E2 3F F7 E2 86-CD C0 ED 06 41 66 0F B7   ....?.......Af..
1273:0290   C9 66 F7 E1 66 89 46 F8-83 7E 16 00 75 38 83 7E   .f..f.F..~..u8.~
1273:02A0   2A 00 77 32 66 8B 46 1C-66 83 C0 0C BB 00 80 B9   *.w2f.F.f.......
1273:02B0   01 00 E8 2B 00 E9 48 03-A0 FA 7D B4 7D 8B F0 AC   ...+..H...}.}...
1273:02C0   84 C0 74 17 3C FF 74 09-B4 0E BB 07 00 CD 10 EB   ..t.<.t.........
1273:02D0   EE A0 FB 7D EB E5 A0 F9-7D EB E0 98 CD 16 CD 19   ...}....}.......
1273:02E0   66 60 66 3B 46 F8 0F 82-4A 00 66 6A 00 66 50 06   f`f;F...J.fj.fP.
1273:02F0   53 66 68 10 00 01 00 80-7E 02 00 0F 85 20 00 B4   Sfh.....~.... ..
-d
1273:0300   41 BB AA 55 8A 56 40 CD-13 0F 82 1C 00 81 FB 55   A..U.V@........U
1273:0310   AA 0F 85 14 00 F6 C1 01-0F 84 0D 00 FE 46 02 B4   .............F..
1273:0320   42 8A 56 40 8B F4 CD 13-B0 F9 66 58 66 58 66 58   B.V@......fXfXfX
1273:0330   66 58 EB 2A 66 33 D2 66-0F B7 4E 18 66 F7 F1 FE   fX.*f3.f..N.f...
1273:0340   C2 8A CA 66 8B D0 66 C1-EA 10 F7 76 1A 86 D6 8A   ...f..f....v....
1273:0350   56 40 8A E8 C0 E4 06 0A-CC B8 01 02 CD 13 66 61   V@............fa
1273:0360   0F 82 54 FF 81 C3 00 02-66 40 49 0F 85 71 FF C3   ..T.....f@I..q..
```

```
1273:0370  4E 54 4C 44 52 20 20 20-20 20 20 00 00 00 00 00    NTLDR      .....
-d
1273:0380  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
1273:0390  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
1273:03A0  00 00 00 00 00 00 00 00-00 00 00 00 0D 0A 4E 54   ..............NT
1273:03B0  4C 44 52 20 69 73 20 6D-69 73 73 69 6E 67 FF 0D   LDR is missing..
1273:03C0  0A 44 69 73 6B 20 65 72-72 6F 72 FF 0D 0A 50 72   .Disk error...Pr
1273:03D0  65 73 73 20 61 6E 79 20-6B 65 79 20 74 6F 20 72   ess any key to r
1273:03E0  65 73 74 61 72 74 0D 0A-00 00 00 00 00 00 00 00   estart..........
1273:03F0  00 00 00 00 00 00 00 00-00 AC BF CC 00 00 55 AA   ..............U.
-
```

对比一下 Debug 调出的数据与图 7-7 的数据，可见完全一样。

可以用相似的方法读写物理盘。只要把上面程序的.data 段中的 FileName 和 Caption 更改成如下：

```
FileName db '\\.\PHYSICALDRIVE0',0
  ......
Caption       db 'NT 中读写物理磁盘',0
```

即可读出机器中第一个硬盘的主引导记录（0 面 0 道 1 扇区），如图 7-9 所示。读者可以自己用 Debug 调出主引导记录出来对比一下结果。

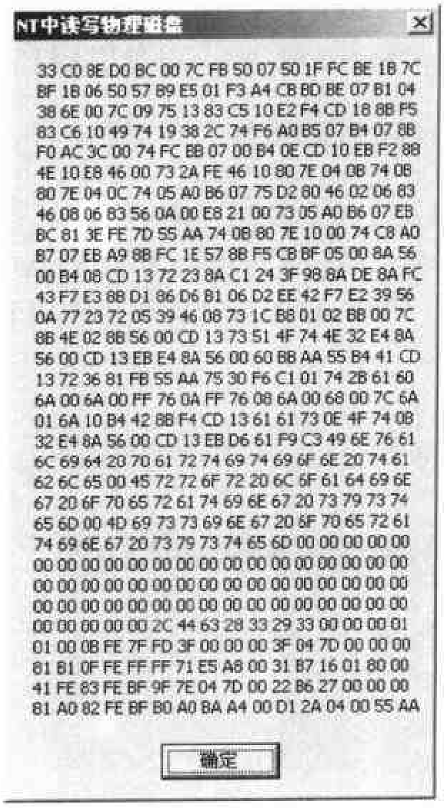


图 7-9

7.4 Windows 9X 下绝对磁盘的读写

前面说过 Windows 95/98 不支持直接用 CreateFile 打开磁盘，因此不能用上节的方法来进行绝对磁盘的读写，而 Windows 9X 本身也没有提供直接读写磁盘的函数。

7.4.1 在 Win32 程序中调用 DOS 功能

Windows 9X 的系统部件 Vwin32 提供了一种让用户在 Win32 应用程序中调用 DOS 中断的方法，即用 DeviceIoControl 向其发控制码和参数。调用 DOS 函数的控制码一般可使用以下几个：

```
VWIN32_DIOC_DOS_DRIVEINFO = 6
VWIN32_DIOC_DOS_INT13   = 4
VWIN32_DIOC_DOS_INT25   = 2
VWIN32_DIOC_DOS_INT26   = 3
VWIN32_DIOC_DOS_IOCTL   = 1
```

这些常量在 Visual C++提供的头文件中是找不到的，但在 MSDN 中是可以查找到这些信息，在 DDK 的头文件 Vwin32.h 和 Vwin32.inc 也有定义。这里对用户要在 Windows 9X 中进行绝对磁盘读写有作用的是 VWIN32_DIOC_DOS_INT25、VWIN32_DIOC_DOS_INT26 和 VWIN32_DIOC_DOS_DRIVEINFO。

顾名思义，VWIN32_DIOC_DOS_INT25 和 VWIN32_DIOC_DOS_INT26 就是调用 DOS 绝对磁盘读写功能 Int 25H 和 Int 26H。但用这两个功能却有问题：它们不支持 FAT32。现在的磁盘通常都有几 GB，Windows 98 机器一般都使用 FAT32，因此要另想办法。

7.4.2 DOS 功能调用 Int 21H 的 7305H 功能

另一个选择是使用 VWIN32_DIOC_DOS_DRIVEINFO。在 Windows 95/98 的 DOS 功能调用 Int 21H 中提供了一个新的磁盘读写功能调用，功能号是 7305H，它支持 FAT32、FAT16 和 FAT12 的读写，使用 VWIN32_DIOC_DOS_DRIVEINFO 则可以调用这个功能。

7305H 功能的参数如下：

AX——功能号 7305H

BX——读写扇区的信息结构

CX——必须是-1

DL——驱动器号，1 是 A 盘，2 是 B 盘，3 是 C 盘，依此类推

SI——读/写标识

其中 AX 和 CX 的值是固定的。

BX 寄存器指向一个结构：

```
DISKIO        STRUC
    dwStartSector      DD ?    ;开始扇区
    wSectors           DW ?    ;扇区数
```

```
    lpBuffer        DD ?    ;缓冲区
ENDS
```

寄存器 SI 中放的是读/写的标识，当 SI 最低位为 0 时表示是读操作，为 1 时表示是写操作。当是写操作时，SI 的第 13、14、15 位也有意义，用于表示所写的数据类型，如表 7-1 所示。

表 7-1 当写时 SI 寄存器位的意义

第 15 位	第 14 位	第 13 位	描述
0	0	0	其他/未知
0	0	1	FAT 数据
0	1	0	目录数据
0	1	1	普通文件数据
1	X	X	保留

设置这些位是为了当写压缩盘时可以正确地写，这些标志千万不能设错。

7.4.3 使用 DeviceIoControl 的 VWIN32_DIOC_DOS_DRIVEINFO 功能

Win32 应用程序中用 DeviceIoControl 调用这个功能还需要把寄存器参数的值传给 Vwin32，同时 Vwin32 还需要把执行后的结果送回给 Win32 应用程序。这在 DeviceIoControl 的第 3，第 5 个参数以冲区的形式传递。这两个参数是一个结构：

```
DIOC_REGISTERS    STRUC
reg_EBX   dd ?
reg_EDX   dd ?
reg_ECX   dd ?
reg_EAX   dd ?
reg_EDI   dd ?
reg_ESI   dd ?
reg_Flags dd ?
ENDS
```

它的意义很直观，就是 6 个通用寄存器和一个标志寄存器。

要调用这个功能，还要先用 CreateFile 取得 Vwin32 的句柄，用于 DeviceIoControl 的第一个参数。

```
Buffer    db 4096 dup (0)
hVxD      dd 0      ;Vwin32 的句柄
drs_in    DIOC_REGISTERS<>
drs_out   DIOC_REGISTERS<>
dio       DISKIO<>
readed    dd 0      ;共读了几个字节
……
```

```
......                  ;填几个结构参数的值
call DeviceIoControl,hVxD,VWIN32_DIOC_DOS_DRIVEINFO,\
                        offset drs_in,size drs_in,\
                        offset drs_out,size drs_out,    \
                        offset readed,0;
```

7.4.4 例程

下面是使用这个功能读逻辑盘的例子。这个程序读 C 盘的引导记录，并用 MessageBox 显示出来。

```
;*****************************
;文件：AbsDisk9X.ASM          *
;功能：Windows 9X 中的绝对    *
;              磁盘存取       *
;*****************************
.386p
.model flat,stdcall   ;平坦内存模式
;外部过程声明
extrn MessageBoxA:proc
extrn ExitProcess:proc
extrn CreateFileA:proc
extrn DeviceIoControl:proc
extrn CloseHandle:proc

;常量定义
NULL   = 0
MB_OK = 0
INVALID_HANDLE_VALUE = -1
FILE_FLAG_DELETE_ON_CLOSE    =    04000000H

VWIN32_DIOC_DOS_DRIVEINFO = 6

DIOC_REGISTERS    STRUC
reg_EBX   dd ?
reg_EDX   dd ?
reg_ECX   dd ?
reg_EAX   dd ?
reg_EDI   dd ?
```

```
reg_ESI    dd ?
reg_Flags    dd ?
ENDS

DISKIO  STRUC
    dwStartSector DD ?    ;开始扇区
    wSectors      DW ?    ;扇区数
    lpBuffer      DD ?    ;缓冲区
ENDS

.data
    FileName db '\\.\vwin32',0
    align 4                 ;双字对齐
    Buffer  db 4096 dup (0)
    hFile    dd 0
    drs_in   DIOC_REGISTERS<>
    drs_out  DIOC_REGISTERS<>
    dio      DISKIO<>
    readed   dd 0

    Caption     db 'Windows 9X 绝对磁盘读写',0
    ErrCreate db '建文件错!',0
    ErrRead     db '读盘错误！ ',0

    ShowTextdb 4096*3 dup (0)
    Digit         db '0123456789ABCDEF'

.code
main proc
    ;取 Vwin32 句柄
    call CreateFileA,offset FileName,0, 0, NULL, 0, FILE_FLAG_DELETE_ON_CLOSE,
NULL
    mov   [hFile],eax
    cmp   eax,INVALID_HANDLE_VALUE
    jnz   read
    ;显示错误信息
    call ShowError,offset ErrCreate
read:      ;读数据
```

```
        mov drs_in.reg_EAX,7305H
        mov drs_in.reg_EBX,offset dio
        mov drs_in.reg_ECX,-1
        mov drs_in.reg_EDX,3  ; 1-A 盘、2-B 盘、3-C 盘，依此类推
        mov drs_in.reg_ESI,0    ;读

        mov dio.dwStartSector,0
        mov dio.wSectors,1
        mov dio.lpBuffer,offset Buffer

        call DeviceIoControl,hFile,VWIN32_DIOC_DOS_DRIVEINFO,\      ;读磁盘
                                offset drs_in,size drs_in,\
                                offset drs_out,size drs_out,    \
                                offset readed,0;
        cmp    eax,0
        jnz    show
        ;显示错误信息
        call ShowError,offset ErrRead
show:       ;显示读出的内容
        mov readed,512
        call ShowBuffer
        ;关闭文件句柄
        call CloseHandle ,[hFile]
        ;退出
        call ExitProcess,0
main endp

ShowError proc ,MESSAGE:DWORD        ;显示出错信息并退出
        call MessageBoxA,NULL,MESSAGE,offset Caption,MB_OK
        cmp    [hFile],0
        jz     endShowErr
        call CloseHandle ,[hFile]        ;关闭句柄
endShowErr:
        call ExitProcess,0               ;退出
ShowError endp

ShowBuffer proc                          ;显示所读出的信息
        ;把数据转换成 16 进制的形式
```

```
        mov   esi,offset Buffer       ;数据
        mov   edi,offset ShowText     ;转换后的数据
        mov   ebx,offset Digit
        mov   ecx,0
        xor   eax,eax

computeAgain:
        cmp   [readed],0
        jz    endCompute
        dec   [readed]
        mov   al,[esi]
        push eax
        shr   eax,4                   ;高 4 位
        mov   al,[ebx+eax]
        mov   byte ptr[edi],al
        inc   edi
        pop   eax
        and   eax,0fH                 ;低 4 位
        mov   al,[ebx+eax]
        mov   byte ptr[edi],al
        inc   edi
        mov   byte ptr[edi],' '       ;空格
        inc   edi
        inc   esi
        inc   ecx
        cmp   ecx,16
        jnz   computeAgain
        xor   ecx,ecx
        mov   byte ptr[edi-1],13      ;回车
        jmp   computeAgain

endCompute:
        ;显示
        call MessageBoxA,NULL,offset ShowText,offset Caption,MB_OK
        ret
ShowBuffer endp

end main
```

如图 7-10 所示是程序的运行情况，你可以对照图 7-7 看这个结果。

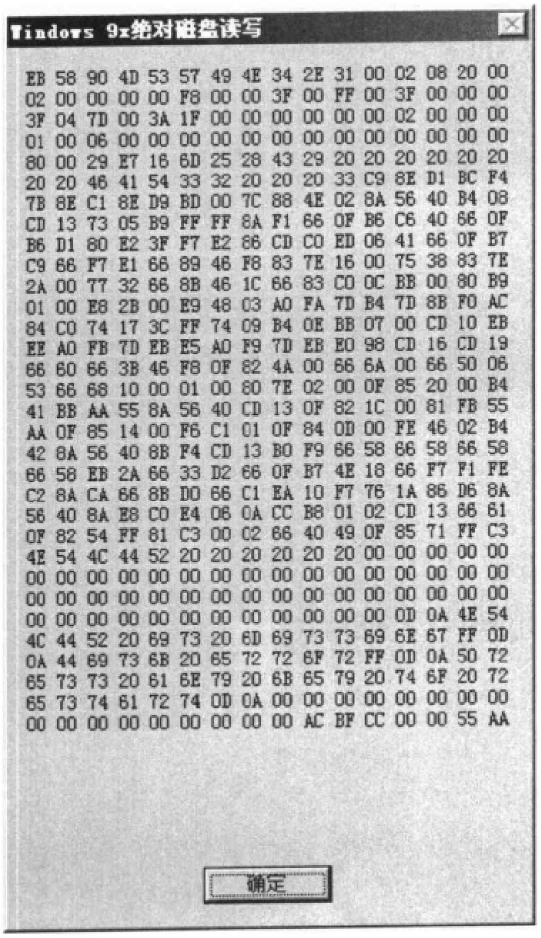


图 7-10

如果要写磁盘，则把 SI 的低位设成 1，并设置适当的数据类型，如：

```
……
mov drs_in.reg_ESI,0    ;写普通文件数据
```

还有一点需注意是在 Windows 95 的零售版中，VWIN32_DIOC_DOS_DRIVEINFO 控制码功能是不可用的。这时可以使用控制码 VWIN32_DIOC_DOS_INT25 和 VWIN32_DIOC_DOS_INT26 来完成，这样，你必须判断操作系统版本的磁盘的格式类型。

7.4.5 Windows 95 中读写物理磁盘

用 VWIN32_DIOC_DOS_INT13 控制码可以直接对物理磁盘进行读写，但这种方法有一个问题，就是对硬盘无效。

使用这个控制码也需要 DIOC_REGISTERS 结构，但因为 Int 13 功能读软盘的参数只需通过寄存器来传送，因此这里不需要 DISKIO 结构作为参数。

这个控制码的使用方法是在将 Int 13 调用所需的参数写进一个 DIOC_REGISTERS 结构中，然后用 VWIN32_DIOC_DOS_INT13 和 Vwin32 句柄及 DIOC_REGISTERS 结构的地址来调用 DeviceIoControl。

下面给出读软盘引导记录的程序。

```
;****************************
;文件：PhyDisk9X.ASM          *
;功能：Windows 9X 下读物理盘 *
;              硬盘不适用      *
;****************************
.386p
.model flat,stdcall   ;平坦内存模式
;外部过程声明
extrn MessageBoxA:proc
extrn ExitProcess:proc
extrn CreateFileA:proc
extrn DeviceIoControl:proc
extrn CloseHandle:proc

;常量定义
NULL   = 0
MB_OK = 0
INVALID_HANDLE_VALUE = -1
FILE_FLAG_DELETE_ON_CLOSE      =     04000000H

VWIN32_DIOC_DOS_INT13 = 4

DIOC_REGISTERS     STRUC
     reg_EBX   dd ?
     reg_EDX   dd ?
     reg_ECX   dd ?
     reg_EAX   dd ?
     reg_EDI   dd ?
     reg_ESI   dd ?
     reg_Flags   dd ?
ENDS

.data
     FileName db '\\.\vwin32',0
     align 4                    ;双字对齐
     Buffer   db 4096 dup (0)
     hFile      dd 0
```

```
        drs_in      DIOC_REGISTERS<>
        drs_out     DIOC_REGISTERS<>
        readed      dd 0

        Caption         db 'Windows 9X 中读写物理磁盘',0
        ErrCreate db '建文件错!',0
        ErrRead         db '读盘错误！ ',0

        ShowTextdb 4096*3 dup (0)
        Digit           db '0123456789ABCDEF'

    .code
    main proc
        ;取 vwin32 句柄
        call CreateFileA,offset FileName,0, 0, NULL, 0, FILE_FLAG_DELETE_ON_CLOSE,
NULL
        mov   [hFile],eax
        cmp   eax,INVALID_HANDLE_VALUE
        jnz   read
        ;显示错误信息
        call ShowError,offset ErrCreate
    read:                   ;读数据
        mov drs_in.reg_EAX,201H         ;Int 13 参数
        mov drs_in.reg_EBX,offset Buffer
        mov drs_in.reg_ECX,1
        mov drs_in.reg_EDX,0            ;只能是：0-A 盘，1-B 盘

        call DeviceIoControl,hFile,VWIN32_DIOC_DOS_INT13,\
                              offset drs_in,size drs_in,\
                              offset drs_out,size drs_out,    \
                              offset readed,0;
        cmp   eax,0
        jnz   show
        ;显示错误信息
        call ShowError,offset ErrRead
    show:    ;显示读出的内容
        mov readed,512
        call ShowBuffer
```

```
        ;关闭文件句柄
        call CloseHandle ,[hFile]
        ;退出
        call ExitProcess,0
main endp

ShowError proc ,MESSAGE:DWORD          ;显示出错信息并退出
        ……;同上
endShowErr:
        call ExitProcess,0          ;退出
ShowError endp

ShowBuffer proc                  ;显示所读出的信息
        ;把数据转换成 16 进制的形式
        ……;同上
ShowBuffer endp

end main
```

程序运行的结果如图 7-11 所示，这是软盘的引导记录。

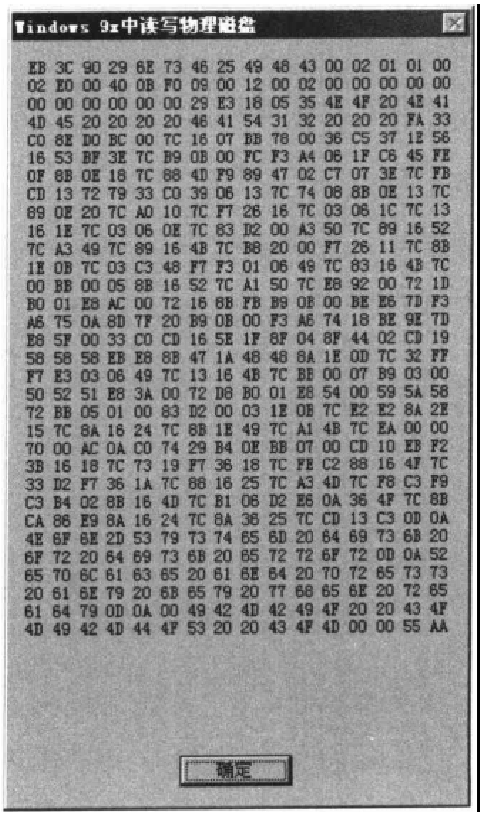


图 7-11

这种方法的缺点是不能操作硬盘，要想操作物理硬盘，一般有两种思路可用，但做起来都不容易。

一是写 16 位代码，如写一个 16 位的 DLL，在这个 DLL 中进行读写操作，然后在用户的 32 程序中载入这个 DLL，并调用其中的函数。但要在 32 位应用程序中调用 16 位的 DLL 是比较麻烦。

另一个思路是写 VxD，并在 Win32 程序中用 DeviceIoControl 发控制码给用户的 VxD，VxD 把 CPU 切入虚拟 86 模式，然后执行 Int 13 来取得 Win32 程序所需的数据，再把 CPU 退出虚拟 86 模式，把数据返回给 Win32 程序。写 VxD 也是比较麻烦，如果考虑不周到就很容易造成系统崩溃。

第八章　图形及界面编程

本章主要介绍 Windows 图形界面接口的编写，包括图形接口 API 理论，菜单设计，子窗口设计，DirectX 应用，遍历窗口设计等。

8.1　图形接口 API

Windows 的图形用户接口 API 是 Windows 系统的重要组成部分，它们为用户提供直观的和美观的界面。这些 API 通常由 Windows 的两个核心组件 GDI32.DLL 和 USER32.DLL 提供。可以说，任何用户友好的 Windows 程序都必定用到它们的 API。

Windows 图形设备接口 API 为程序员提供各种便于使用的编程接口，这些 API 与设备无关。也就是说，不管所操纵的设备对象是什么，都可以使用同一个 API。例如，不管是要在屏幕上还是在打印机上画一个椭圆，都可以调用 Ellipse API 来完成，而不需考虑显示器与打印机有什么不同。

GDI 的 API 所提供的方法很丰富，包括文字显示、各种曲线和标准图形（如直线、椭圆、矩形、贝塞尔曲线等）、以及各种图形操作等。GDI32.DLL 的 API 通常都要与 USER32.DLL 的 API 联合使用。典型的应用是：在调用 GDI 的 API 时，一般都有一个“设备上下文”名柄的参数，这个句柄在 Windows 内部用于标识具体设备的数据结构，为了得到这个句柄，就需要调用 USER32 的 API GetDC，用完后又要调用 ReleaseDC 来释放这个句柄。

下面的例子演示 GDI API 的使用，如图 8-1 所示。

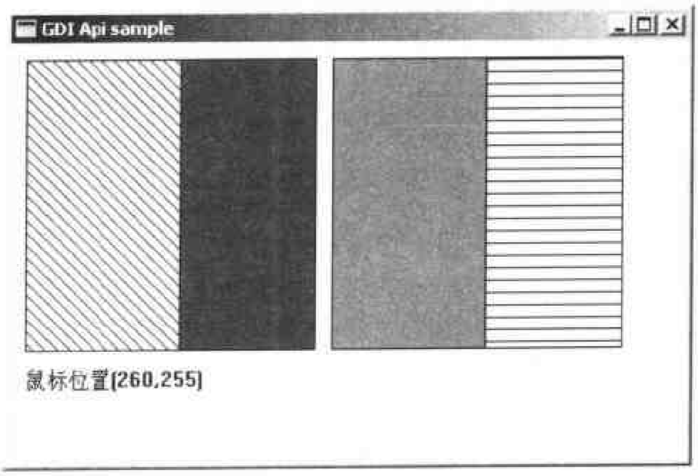


图 8-1

这个程序的功能是用不同的画笔在窗体上画矩形，并在鼠标在窗体上移动时写出鼠标的位置。

这个程序用到了 GDI 编程时用到的几个常用 API：

1）BeginPaint 和 EndPaint，当消息处理函数收到 WM_PAINT 消息时，需要使用这两

个 API，并且它只应在处理 WM_PAINT 消息时使用。它的调用方式如下：

```
PAINTSTRUCT struc
    hdc  dd ?
    fErase    dd  ?
    rcPaint   RECT<>
    fRestore  dd  ?
    fIncUpdate    dd  ?
    rgbReserverd  32 dup (?)
        ends
    ps   PAINTSTRUCT<>
……
call  BeginPaint,handle,offset ps
mov       hdc,eax
;做输出工作……
call  EndPaint,handle,offset ps;
```

结构 PAINTSTRUCT 由 BeginPaint 填充，它的前 3 个域由用户使用，后 3 个参数由系统使用，用户程序不应改动它。

域 hdc 是得到“设备上下文”的句柄。所谓“设备上下文”即是描述设备环境属性的数据结构；域 fErase 是表明是否需要重画；域 rcPaint 是一个矩形区域，表示要重画的区域坐标。

2）GetDC 和 ReleaseDC，当你不是在 WM_PAINT 消息中使用“设备上下文”时，需要这 2 个 API。它们取得和释放“设备上下文”。

3）CreateHatchBrush，用指定的图案和颜色建立一个逻辑画笔。

4）CreateSolidBrush，用指定颜色建立一个逻辑画笔。

5）SelectObject，把一个对象如画笔、字体等选入指定的“设备上下文”。

6）Rectangle，画一个矩形。

这个程序的代码如下：

```
;**********************************
;文件：GDIAPI.asm                 *
;功能：使用 GDI 的 API 的例子      *
;**********************************
.386p
.model flat,stdcall
include win32.inc
extrn GetMessageA:proc
extrn TranslateMessage:proc
extrn DispatchMessageA:proc
extrn ShowWindow:proc
```

```
extrn UpdateWindow:proc
extrn CreateWindowExA:proc
extrn RegisterClassA:proc
extrn DefWindowProcA:proc
extrn MessageBoxA:proc
extrn ExitProcess:proc
extrn LoadIconA:proc
extrn LoadCursorA:proc
extrn GetStockObject:proc
extrn GetModuleHandleA:proc
extrn PostQuitMessage:proc
extrn _wsprintfA:proc
extrn SetWindowTextA:proc
extrn Rectangle:proc
extrn EndPaint:proc
extrn SelectObject:proc
extrn DeleteObject:proc
extrn BeginPaint:proc
extrn CreateHatchBrush:proc
extrn CreateSolidBrush:proc
extrn CreatePenIndirect:proc
extrn GetDC:proc
extrn TextOutA:proc
extrn ReleaseDC:proc

POINT struc
    x dd ?
    y dd ?
    ends

MSG struc
    hwnd      dd ?
    message   dd ?
    wParam    dd ?
    lParam    dd ?
    time dd ?
    pt        POINT<>
    ends
```

```
WNDCLASS struc
     style       dd ?
     lpfnWndProc  dd ?
     cbClsExtra    dd ?
     cbWndExtra   dd ?
     hInstance dd ?
     hIcon          dd ?
     hCursor        dd ?
     hbrBackground      dd ?
     lpszMenuName      dd ?
     lpszClassName      dd ?
     ends

.data
     myclassname  db 'MyClass',0
     Caption          db 'GDI Api sample',0
     string             db   30 dup (?)
     stringTmp       db '鼠标位置(%d,%d)        ',0
     align 4
     hBrsh0    dd ?
     hBrsh1    dd ?
     hBrsh2    dd ?
     hBrsh3    dd ?
     hdc  dd ?
     hInstdd ?
     hWnd       dd ?
     ps    PAINTSTRUCT<>
     msg MSG<>
     wc   WNDCLASS<>

.code
main:
     call  GetModuleHandleA,0
     mov [hInst],eax

     mov wc.lpszClassName,offset myclassname
     mov eax,[hInst]
     mov wc.hInstance,eax
```

```
        mov wc.lpfnWndProc,offset WinMsgProc
        call  LoadIconA,0,IDI_APPLICATION
        mov wc.hIcon,eax
        call  LoadCursorA,0,IDC_IBEAM
        mov wc.hCursor,eax
        mov wc.lpszMenuName,0
        call  GetStockObject,WHITE_BRUSH
        mov wc.hbrBackground,eax
        mov wc.style,CS_HREDRAW or CS_VREDRAW or CS_GLOBALCLASS
        mov wc.cbClsExtra,0
        mov wc.cbWndExtra,0

        call  RegisterClassA,offset wc              ;注册窗口类
        ;建立窗口
        call  CreateWindowExA, 0, offset myclassname, offset Caption,
WS_OVERLAPPEDWINDOW, 100,100,450,300,0,0,[hInst],0
        mov [hWnd],eax

        call  ShowWindow,[hWnd],1
        call  UpdateWindow,[hWnd]
    MsgLoop:                            ;消息循环
        call  GetMessageA,offset msg,0,0,0
        or    eax,eax
        jz    Exit
        call  TranslateMessage,offset msg
        call  DispatchMessageA,offset msg
        jmp MsgLoop
    Exit:
        call ExitProcess,0
    ;*******************************************************
    ;消息处理函数
    WinMsgProc    proc  uses  ebx  edi  esi,  handle:DWORD,  wMsg:DWORD,
wParam1:DWORD, lParam1:DWORD
        cmp wMsg,WM_CREATE
        jz    create
        cmp wMsg,WM_DESTROY
        jz    destory
        cmp wMsg,WM_MOUSEMOVE
```

```
        jz    mousemove
        cmp wMsg,WM_PAINT
        jz    paint
        jmp default
create:
        xor  eax,eax
        ret
destory:
        call  PostQuitMessage,0
        xor  eax,eax
        ret
mousemove:                  ;鼠标移动，显示鼠标位置
        mov eax,lParam1
        movzx     ebx,ax
        shr  eax,16
        call  _wsprintfA,offset string,offset stringTmp,ebx,eax
        push eax
        call  GetDC,handle
        mov hdc,eax
        call  TextOutA,hdc,10,210,offset string
        call  ReleaseDC,handle,hdc
        add  esp,4*4
        xor  eax,eax
        ret
paint:                      ;画矩形
        ;建立画刷
        call  CreateHatchBrush,HS_FDIAGONAL,000000
        mov hBrsh0,eax
        call  CreateSolidBrush,00000ffH
        mov hBrsh1,eax
        call  CreateSolidBrush, 00ff00H
        mov hBrsh2,eax
        call  CreateHatchBrush,HS_HORIZONTAL,0ff0000H
        mov hBrsh3,eax

        call  BeginPaint,handle,offset ps
        mov       hdc,eax
        ;画矩形
```

```
    call  SelectObject,hdc,hBrsh0
    call  Rectangle,hdc,10,10.200,200
    call  SelectObject,hdc,hBrsh1
    call  Rectangle,hdc,110,10,200,200
    call  SelectObject,hdc,hBrsh2
    call  Rectangle,hdc,210,10,400,200
    call  SelectObject,hdc,hBrsh3
    call  Rectangle,hdc,310,10,400,200

    call  EndPaint,handle,offset ps
    ;删除画刷
    call  DeleteObject,hBrsh0
    call  DeleteObject,hBrsh1
    call  DeleteObject,hBrsh2
    call  DeleteObject,hBrsh3
    xor   eax,eax
    ret
default:
    call  DefWindowProcA,handle,wMsg,wParam1,lParam1
    ret
WinMsgProc    endp
    end main
```

8.2 菜单

菜单的发明可以说是程序易用性方面的伟大革命，若是写了一个很多功能的程序而没有使用菜单，那几乎就可以断定说，这个程序不会受欢迎。

菜单可分为两种：一种固定在窗口上，另一种是弹出式的快捷菜单。菜单是一种资源，一般在资源文件（.rc 文件）中定义，但也可以动态生成。一个包含菜单资源的资源文件例子如下：

```
#define IDM_HELLO         0x200
#define IDM_ABOUT         0x201
#define IDM_TOP           0x202
#define IDM_POPUP         0x203
#define IDM_EXIT          0X204

MyFirstMenu MENU DISCARDABLE
{
```

```
        POPUP "Files"
        {
            MENUITEM "Hello",                   IDM_HELLO
            MENUITEM SEPARATOR
            MENUITEM "Exit",                    IDM_EXIT
        }
        POPUP "Help"
        {
            MENUITEM "关于...",                  IDM_ABOUT
        }
        MENUITEM "顶层",                         IDM_TOP
    }

    MySecondMenu MENU DISCARDABLE
    {
        POPUP "Files"
        {
            MENUITEM "弹出菜单",                 IDM_POPUP
            MENUITEM SEPARATOR
            MENUITEM "退出",                     IDM_EXIT
        }
    }
```

要使用固定的菜单，只需在注册 Window 类时，把 WNDCLASS 的 lpszMenuName 成员指向资源中的菜单名字即可：

```
MenuName    db 'MyFirstMenu',0
......
mov wc.lpszMenuName,offset MenuName
```

要显示弹出式的快捷菜单，应使用 TrackPopupMenu 函数：

```
call    TrackPopupMenu,hmenu,flages,x,y,0,hwin,0
```

其中：

1）最后一个和倒数第三个参数必须为 0。

2）hmenu 是菜单的名柄。

3）f lages 是设置显示时的鼠标对齐位置和及其它一些属性，一般置为 TPM_LEFTALIGN 即可。

4）x,y 是坐标。

5）hWin 是快捷菜单的属主。

Hmenu 参数所指的菜单可以用 LoadMenuA 和 GetSubMenu 从资源文件中取出：

```
Call    LoadMenuA,hInst,offset MenuName
```

```
Call    GetSubMenu,eax,pos
mov hmenu,eax
```

其中：

1）hInst 是拥有菜单资源的模块实例。

2）MenuName 是菜单名称。

3）GetSubMenu 的第 2 个参数 pos 是子菜单在菜单中的位置，由 0 数起。

菜单也可以自己用 CreatePopupMenu 和 AppendMenuA 动态生成：

```
call  CreatePopupMenu
mov hmenu,eax
call  AppendMenuA,hmenu,uFlags,id,offset data
```

其中：

1）uFlages 是表明菜单项是字符串、位图等。

2）id 菜单的标识符。

3）data 是菜单的字符串指针或位图指针等。

下面的例子示范如何在程序中使用固定的菜单和弹出的菜单，以及如何动态生成菜单，如图 8-2 所示。

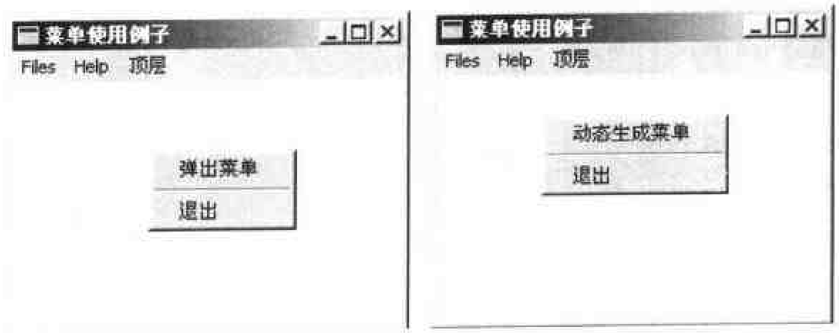


图 8-2

```
;*********************************
;文件：Menu.asm                   *
;功能：演示使用菜单、弹出式菜      *
;          单和动态生成菜单         *
;*********************************

.386p
.model flat,stdcall
include win32.inc
extrn GetMessageA:proc
extrn TranslateMessage:proc
extrn DispatchMessageA:proc
extrn ShowWindow:proc
```

```
extrn UpdateWindow:proc
extrn CreateWindowExA:proc
extrn RegisterClassA:proc
extrn DefWindowProcA:proc
extrn MessageBoxA:proc
extrn ExitProcess:proc
extrn LoadIconA:proc
extrn LoadCursorA:proc
extrn GetStockObject:proc
extrn GetModuleHandleA:proc
extrn PostQuitMessage:proc
extrn CreatePopupMenu:proc
extrn InsertMenuItemA:proc
extrn GetCursorPos:proc
extrn TrackPopupMenu:proc
extrn GetMenu:proc
extrn GetSubMenu:proc
extrn LoadMenuA:proc
extrn SetMenuItemInfoA:proc
extrn AppendMenuA:proc
extrn DestroyMenu:proc
extrn _wsprintfA:proc

FALSE = 0
MB_OK = 0
NULL   = 0

IDM_HELLO          = 200H
IDM_ABOUT          = 201H
IDM_TOP            = 202H
IDM_POPUP          = 203H
IDM_EXIT           = 204H
IDM_DYNAMIC        = 205H
MF_STRING         = 00000000H
MF_SEPARATOR      = 00000800H
MFT_STRING        = MF_STRING
MFT_SEPARATOR     = MF_SEPARATOR
TPM_LEFTALIGN     = 0000H
```

```
POINT struc
    x dd ?
    y dd ?
    ends
MSG struc
    hwnd      dd ?
    message   dd ?
    wParam    dd ?
    lParam    dd ?
    time dd ?
    pt        POINT<>
    ends

WNDCLASS struc
    style          dd ?
    lpfnWndProc    dd ?
    cbClsExtra     dd ?
    cbWndExtra     dd ?
    hInstance dd ?
    hIcon          dd ?
    hCursor        dd ?
    hbrBackground      dd ?
    lpszMenuName       dd ?
    lpszClassName      dd ?
    ends

MENUITEMINFO struc
    cbSize     dd ?
    fMask      dd ?
    fType      dd ?
    fState     dd ?
    wID        dd ?
    hSubMenu           dd ?
    hbmpChecked        dd ?
    hbmpUnchecked      dd ?
    dwItemData         dd ?
    dwTypeData         dd ?
    cch                dd ?
```

```
    ends

.data
    printMsgTitle       db 'title',0
    printMsgTmp         db 'HWND:%X MSG:%X WPARAM:%X LPARAM:%X',0
    printMsgText        db 200 dup (0)
    MenuName            db 'MyFirstMenu',0
    MenuName1           db 'MySecondMenu',0
    myclassname         db 'MyClass',0
    caption             db '菜单使用例子',0
    hello               db 'Hello',0
    about               db '菜单例子 Ver 1.0',13,' 蓝琚成,2000/8',0
    top                 db '顶层',0
    popup               db '弹出菜单例子',0
    exit                db '退出',0
    dynamic             db '动态生成菜单',0
    align 4
    hInst dd ?
    hWnd     dd ?
    hmenu    dd ?
    hmenu1   dd ?
    hmenu0   dd ?
    msg      MSG<>
    wc       WNDCLASS<>
    point    POINT<>
    menuinfo MENUITEMINFO<>

.code
main:
    call  GetModuleHandleA,0          ;取模块句柄
    mov [hInst],eax

    mov wc.lpszClassName,offset myclassname
    mov eax,[hInst]
    mov wc.hInstance,eax
    mov wc.lpfnWndProc,offset WinMsgProc
    call  LoadIconA,0,IDI_APPLICATION
    mov wc.hIcon,eax
```

```
        call  LoadCursorA,0,IDC_IBEAM
        mov wc.hCursor,eax
        mov wc.lpszMenuName,offset MenuName   ;菜单名
        call  GetStockObject,WHITE_BRUSH
        mov wc.hbrBackground,eax
        mov wc.style,CS_HREDRAW or CS_VREDRAW or CS_GLOBALCLASS
        mov wc.cbClsExtra,0
        mov wc.cbWndExtra,0

        call  RegisterClassA,offset wc          ;注册窗口类
        ;建立主窗口
        call  CreateWindowExA,0,offset myclassname,offset caption,WS_OVERLAPPEDWI
NDOW,100,100,450,300,0,0,[hInst],0
        mov [hWnd],eax

        call  ShowWindow,[hWnd],1
        call  UpdateWindow,[hWnd]
    MsgLoop:                            ;消息循环
        call  GetMessageA,offset msg,0,0,0
        or    eax,eax
        jz    Exit
        call  TranslateMessage,offset msg
        call  DispatchMessageA,offset msg
        jmp  MsgLoop
    Exit:
        call  ExitProcess,0
    ;**************************************************************************
    WinMsgProc    proc  uses  ebx  edi  esi,  handle:DWORD,  wMsg:DWORD,
wParam1:DWORD, lParam1:DWORD
        cmp wMsg,WM_CREATE
        jz   create
        cmp wMsg,WM_DESTROY
        jz   destory
        cmp wMsg,WM_LBUTTGNDOWN
        jz   lbuttondown
        cmp wMsg,WM_RBUTTONUP
        jz   rbuttonup
        cmp wMsg,WM_COMMAND
```

```
        jz    command
        jmp  default
command:                        ;处理菜单消息
        mov ebx,wParam1
        cmp bx,IDM_HELLO
        jz    menu_hello
        cmp bx,IDM_ABOUT
        jz    menu_about
        cmp bx,IDM_TOP
        jz    menu_top
        cmp bx,IDM_POPUP
        jz    menu_popup
        cmp bx,IDM_EXIT
        jz    menu_exit
        cmp bx,IDM_DYNAMIC
        jz    menu_dynamic
        call  printMsg,handle,wMsg,wParam1,lParam1
        jmp  default
menu_hello:
        call  MessageBoxA,handle,offset hello,offset caption,MB_OK
        xor   eax,eax
        ret
menu_about:
        call  MessageBoxA,handle,offset about,offset caption,MB_OK
        xor   eax,eax
        ret
menu_top:
        call  MessageBoxA,handle,offset top,offset caption,MB_OK
        xor   eax,eax
        ret
menu_popup:
        call  MessageBoxA,handle,offset popup,offset caption,MB_OK
        xor   eax,eax
        ret
menu_exit:
        jmp  destory
menu_dynamic:
        call  MessageBoxA,handle,offset dynamic,offset caption,MB_OK
```

```
        xor   eax,eax
        ret
create:                ;动态建立弹出菜单
        call  CreatePopupMenu
        mov hmenu,eax
        call  AppendMenuA,hmenu,MF_STRING,IDM_DYNAMIC,offset dynamic
        call  AppendMenuA,hmenu,MFT_SEPARATOR,47H,NULL
        call  AppendMenuA,hmenu,MF_STRING,IDM_EXIT,offset exit
        ;取菜单
        call  LoadMenuA,hInst,offset MenuName1
        mov hmenu0,eax
        call  GetSubMenu,eax,0
        mov hmenu1,eax
        xor   eax,eax
        ret
destory:               ;删除菜单资源
        call  DestroyMenu,hmenu
        call  DestroyMenu,hmenu0
        call  PostQuitMessage,0
        xor   eax,eax
        ret
rbuttonup:             ;弹出程序初始化时动态生成的菜单
        call  GetCursorPos, offset point
        call  TrackPopupMenu,hmenu,TPM_LEFTALIGN,point.x,point.y,0,handle,0
        xor   eax,eax
        ret
lbuttondown:           ;弹出程序初始化时取出的菜单
        call  GetCursorPos, offset point
        call  TrackPopupMenu,hmenu1,TPM_LEFTALIGN,point.x,point.y,0,handle,0
        xor   eax,eax
        ret
default:
        call  DefWindowProcA,handle,wMsg,wParam1,lParam1
        ret
WinMsgProc    endp

printMsg proc aaaa:DWORD,bbbb:DWORD,cccc:DWORD,dddd:DWORD
        call  _wsprintfA,offset printMsgText,offset printMsgTmp,aaaa,bbbb,cccc,dddd
```

```
        add  esp,4*6
        call  MessageBoxA,NULL,offset printMsgText,offset printMsgTitle,MB_OK
        ret
printMsg endp
end main
```

8.3 子窗口

一个实用的程序界面一般都不会只用一个主窗口，通常它还会有子窗口，这些子窗口之间或子窗口与主窗口之间还会相互交换信息。如果用一个 DialogBox 作为主窗口，则可以在资源文件中先定义好子窗口的位置、数目、风格等信息。但是，有时我们也需要在程序中动态地生成子窗口。下面我们给出动态生成子窗口及窗口间相互作用的例子。

Windows 预定义了几种子窗口类，它们也叫“控制”，具体如下：

1）BUTTON，按钮控制。

2）COMBOBOX，组合框控制。

3）EDIT，编辑控制。

4）LISTBOX，列表控制。

5）MDICLIENT，MDI 子窗口。

6）RichEdit，可以作多行编辑的控制。

7）RICHEDIT_CLASS，RichEdit 的 2.0 版。

8）SCROLLBAR，滚动条控制。

9）STATIC，静态控制，用以显示文字。

生成子窗口与生成主窗口一样，都使用 CreateWindowExA 函数，但是，在风格中要使用 WS_CHILD 风格。

下面的例子在建立时生成 4 个子窗口，如图 8-3 所示。它分别使用了 STATIC 类、EDIT 类、BUTTON 类和 LISTBOX 类。

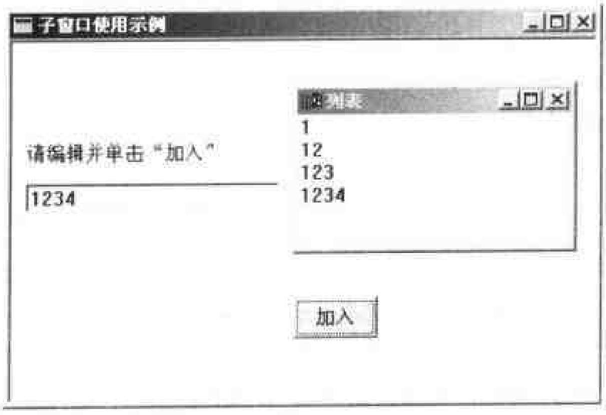


图 8-3

程序的运作过程是用户在 EDIT 控制中写入字符串，按“加入”按钮，则 EDIT 控制中的内容被加入 ListBox 中，ListBox 的项数加 1。当加到一定数目时，再加入一个后会自

动删除最旧的一个。

向 ListBox 中增加或删除一项，只要向其发送 LB_ADDSTRING 或 LB_DELETESTRING 消息即可。

```
;*****************************
;文件: SubWin.asm              *
;功能: 动态生成及使用子窗口     *
;*****************************

.386p
.model flat,stdcall
include win32.inc
extrn GetMessageA:proc
extrn TranslateMessage:proc
extrn DispatchMessageA:proc
extrn ShowWindow:proc
extrn UpdateWindow:proc
extrn CreateWindowExA:proc
extrn RegisterClassA:proc
extrn DefWindowProcA:proc
extrn MessageBoxA:proc
extrn ExitProcess:proc
extrn LoadIconA:proc
extrn LoadCursorA:proc
extrn GetStockObject:proc
extrn GetModuleHandleA:proc
extrn PostQuitMessage:proc
extrn SendMessageA:proc
extrn GetWindowTextA:proc

NULL    =0
WS_EX_CLIENTEDGE     =    0200h
LB_DELETESTRING      =0182H
LB_ADDSTRING         =0180H

POINT struc
    x dd ?
    y dd ?
    ends
```

```
MSG struc
    hwnd      dd ?
    message   dd ?
    wParam    dd ?
    lParam    dd ?
    time dd ?
    pt        POINT<>
    ends

WNDCLASS struc
    style          dd ?
    lpfnWndProc    dd ?
    cbClsExtra     dd ?
    cbWndExtra     dd ?
    hInstance dd ?
    hIcon          dd ?
    hCursor        dd ?
    hbrBackground       dd ?
    lpszMenuName        dd ?
    lpszClassName       dd ?
    ends

.data
    Text       db 256 dup (0)
    myclassname  db 'MyClass',0
    caption      db '子窗口使用示例',0
    align 4
    hInst        dd ?
    hWnd         dd ?
    hBUTTON      dd ?
    hLISTBGX     dd ?
    hEDIT        dd ?
    cList        dd 0
    msg          MSG<>
    wc           WNDCLASS<>

.code
main:
```

```
        call    GetModuleHandleA,0
        mov     [hInst],eax

        mov     wc.lpszClassName,offset myclassname
        mov     eax,[hInst]
        mov     wc.hInstance,eax
        mov     wc.lpfnWndProc,offset WinMsgProc
        call    LoadIconA,0,IDI_APPLICATION
        mov     wc.hIcon,eax
        call    LoadCursorA,0.IDC_IBEAM
        mov     wc.hCursor,eax
        mov     wc.lpszMenuName,0
        call    GetStockObject,WHITE_BRUSH
        mov     wc.hbrBackground,eax
        mov     wc.style,CS_HREDRAW or CS_VREDRAW or CS_GLOBALCLASS
        mov     wc.cbClsExtra,0
        mov     wc.cbWndExtra,0

        call    RegisterClassA,offset wc            ;注册窗口类
        ;建立窗口
        call    CreateWindowExA, WS_EX_CLIENTEDGE, offset myclassname, offset
caption, WS_OVERLAPPEDWINDOW, 100,100,450,300,0,0,[hInst],0
        mov [hWnd],eax

        call ShowWindow,[hWnd],1
        call UpdateWindow,[hWnd]
    MsgLoop:                        ;消息循环
        call GetMessageA,offset msg,0,0,0
        or   eax,eax
        jz   Exit
        call TranslateMessage,offset msg
        call DispatchMessageA,offset msg
        jmp MsgLoop
    Exit:
        call ExitProcess,0
    ;*************************************************************
    ;消息处理
    WinMsgProc      proc    uses    ebx    edi    esi,    handle:DWORD,    wMsg:DWORD,
```

```
wParam1:DWORD, lParam1:DWORD
        cmp wMsg,WM_CREATE
        jz   create
        cmp wMsg,WM_DESTROY
        jz   destory
        cmp wMsg,WM_COMMAND
        jz   command
        jmp default

   create:
        call  InitSubWin,handle
        xor  eax,eax
        ret

   command:
        mov eax,hBUTTON
        cmp lParam1,eax
        jnz  default
        ;把 Edit 控制中的文字加入 LISTBOX
        call  GetWindowTextA,hEDIT,offset Text,256
        call  SendMessageA,hLISTBOX,LB_ADDSTRING,0,offset Text
        cmp cList,10                ;满 10 条?
        jnb  delitem
        inc  cList
        xor  eax,eax
        ret

   delitem:                   ;删除 1 条
        call  SendMessageA,hLISTBOX,LB_DELETESTRING,0,0
        xor  eax,eax
        ret

   destory:
        call  PostQuitMessage,0
        xor  eax,eax
        ret

   default:
```

```
        call DefWindowProcA,handle,wMsg,wParam1,lParam1
        ret

WinMsgProc    endp
;***********************************************
        szBUTTON      db 'button',0
        szLISTBOX     db 'listbox',0
        szEDIT        db 'EDIT',0
        szSTATIC      db 'STATIC',0
        szBUTTON_TEXT db '加入',0
        szLISTBOX_TEXT      db '列表',0
        szEDIT_TEXT         db '编辑',0
        szSTATIC_TEXT       db '请编辑并单击“加入”',0

;生成子窗口
InitSubWin proc hMain:DWORD
        call CreateWindowExA, 0, offset szLISTBOX, offset szLISTBOX_TEXT, WS_CHILD or WS_OVERLAPPEDWINDOW or WS_VISIBLE or WS_VSCROLL, 200,40,200,120,hMain,0,0,0
        mov hLISTBOX,eax
        call CreateWindowExA, 0, offset szSTATIC , offset szSTATIC_TEXT, WS_CHILD or WS_VISIBLE, 10,70 ,180,20,hMain,0,0,0
        call CreateWindowExA, WS_EX_CLIENTEDGE, offset szEDIT, offset szEDIT_TEXT, WS_CHILD or WS_VISIBLE, 10,100,180,20,hMain,0,0,0
        mov hEDIT,eax
        call CreateWindowExA, 0, offset szBUTTON, offset szBUTTON_TEXT, WS_CHILD or WS_VISIBLE, 200,180,60,30,hMain,0,0,0
        mov hBUTTON,eax
        ret
InitSubWin endp
end main
```

8.4 DirectX

在 DOS 的时代，程序员为取得高速度而直接对硬件编程，这必须要有过硬的硬件知识和丰富的编程经验。在 Windows 中，由于系统的保护和 Windows 系统的高度复杂，要想直接对硬件编程已经越来越难了。但是，不用担心，微软提供的 DirectX 提供了直接访问硬件的方法，而又不需要用户真的去了解硬件的细节，因为这些在 DirectX 中已封装好了。

DirectX 包括有 DirectDraw、DirectSound、DirectPlay、Direct3D 等，其中最常用的是 DirectDraw。它提供接口给程序员直接访问显示硬件及进行各种硬件加速的换页、缩放等操作，使程序尽可能利用硬件的加速性能而不需知道硬件的具体细节。

8.4.1 COM 对象在内存中的表示

用汇编语言写 DirectX 程序最主要的问题是 DirectX 是基于 COM（Component Object Model，组件对象模型）的，COM 与 C++非常类似，事实上，它们在二进制上兼容。这是面向对象的程序设计，用汇编语言编写这种程序时，必须知道对象在内存中是如何表示的。可能你会问，如何取得一个类的成员变量的值？如何取得它的成员函数的地址？

在 C++中，类（class）跟结构（struct）是很相似的。不同的是在结构中，所有成员，包括成员函数和成员变量都默认是公有（public）的。当我们定义一个结构及其实例时，它的实例在内存中占用了与其成员变量大小相等的内存。例如：

```
typedef   struct _ST
{
      int a;
      char *p;
}ST;
ST    st1;
```

则 sizeof(st1)的值为 8，因为类型 int 和 char *的大小都为 4。类与此相似，若把上面的 struct 改为 class，则会得到相同的结果。这时要从 st1 中用汇编语言取出 a 和 p 的值，可以通过下面的指令：

```
mov ebx,offset st1
mov eax,[ebx]            ;取得 a 的值
mov edx,[ebx+4]          ;取得 p 的值
```

在 C++中，类还可能包含一个指向一张“虚函数表”的指针。这张表里包含有该类的所有虚函数的指针。

例如，对于类 A1：

```
class   A1
{
public:
      void AA();
      void BB();
      int a,b;
};
```

类 A2：

```
class   A2
{
```

```
public:
    virtual void AA();
    virtual void BB();
    int a,b;
};
```

则 sizeof(A1)的值是 8，而 sizeof（A2）的值是 12，因为类 A2 多了一个指向虚函数表的指针，如图 8-4 所示。

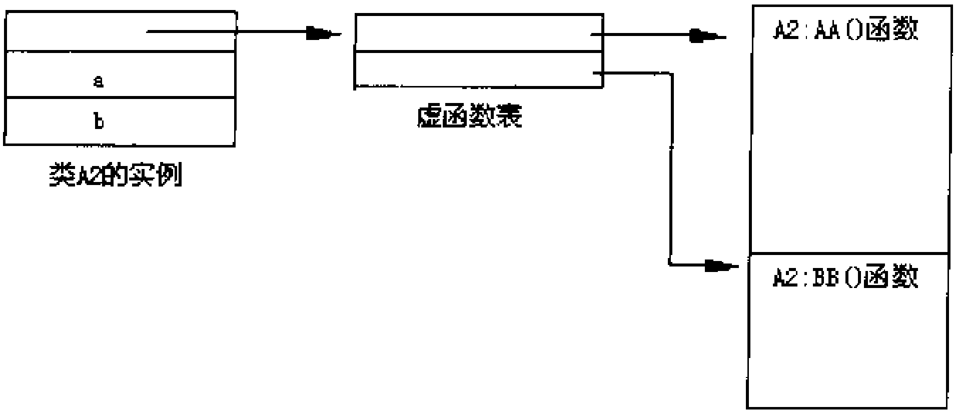


图 8-4

如果我们要用汇编语言取得 A2 的虚函数 BB 的地址，可以通过下面语句（设 a2 是 A2 的实例）：

```
mov ebx,offset a2        ;取得 a2 的地址
mov ebx,[ebx]            ;取得虚函数表地址
mov eax,[ebx+4]          ;取得 A2:BB()的地址
```

C++正是通过这样支持虚特性，即当把一个子类的对象的地址传给父类的指针时，仍可以通过指针取得子类的虚函数的真正地址，而不会调用了父类的虚函数。

另一点是关于 this 指针的问题。当一个被成员函数调用时，它需要一个 this 指针作为参数。this 指针实际上就是对象的内存地址，成员函数通过它来存取成员变量。我们在前面说过，C 的参数从右到左入栈，C++也一样，而它的 this 参数是最后入栈，即相当于在参数列表的最左边加上一个 this 参数。例如在上面的汇编语言段中，将取得 A2:BB()的地址放到 eax 中后，要调用它，则应用下面的语句：

```
mov ebx,offset a2
push ebx
call eax
```

在 Visual C++中，函数返回时，堆栈参数是由被调用者清理的。

需要说明的是，并非所有编译器都是如此处理的，甚至 Visual C++本身也并非总是如此处理。例如，可能会通过约定的寄存器而不是堆栈传递 this 指针，或按 C 而不是 stdcall 标准来调整堆栈，又或者是虚函数表的指针不在对象的开始处，等等，这要看具体而定。但是，COM 因为可在各个编译器上（如 Visual C++，BCB 等）都能编译，因此它是遵守

这些规定的。

DirectX 的类继承自 IUnknown 类，它的定义如下：

```
class Iunknown
{
    virtual HRESULT QueryInterface(REFIID iid,void **ppv);
    virtual ULONG AddRef();
    virtual ULONG Release();
};
```

因此，所有 DirectX 的类都有这三个成员函数。

8.4.2 DirectDraw

DirectDraw 是 DirectX 中用得最为广泛的一种。下面的例子演示如何在汇编语言中使用 DirectDraw。程序先把屏幕分辨率置成 640×480 和 256 色，并在用户按下鼠标时在窗体上显示字和一些简单的图形。

程序开始用 DirectDrawCreate 建立一个 DirectDraw 对象，并用 DirectDraw 对象的成员函数 SetCooperativeLevel 设成全屏和独占模式；用成员函数 SetDisplayMode 把屏幕设成 640×480 和 8 位颜色深度；用成员函数 CreateSurface 生成主一个页面对象，即 DirectDrawSurface 对象。当鼠标单击时，用主页面对象的 GetDC 和 ReleaseDC 成员函数取得和释放“设备上下文”，并用标准的 GDI 函数 TextOut，LintTo 等在窗体上写字和画图。程序退出时释放 DirectDraw 对象和主页面对象。

```
;*********************************
;文件：DirectDraw.asm              *
;功能：演示如何使用 DirectDraw     *
;*********************************
.386p
.model flat,stdcall
include win32.inc
extrn GetMessageA:proc
extrn TranslateMessage:proc
extrn DispatchMessageA:proc
extrn ShowWindow:proc
extrn UpdateWindow:proc
extrn CreateWindowExA:proc
extrn RegisterClassA:proc
extrn DefWindowProcA:proc
extrn MessageBoxA:proc
extrn ExitProcess:proc
```

```
extrn LoadIconA:proc
extrn LoadCursorA:proc
extrn GetStockObject:proc
extrn GetModuleHandleA:proc
extrn PostQuitMessage:proc
extrn GetSystemMetrics:proc
extrn DirectDrawCreate:proc
extrn SetBkColor:proc
extrn SetTextColor:proc
extrn TextOutA:proc
extrn SetTextAlign:proc
extrn MoveToEx:proc
extrn LineTo:proc
extrn Rectangle:proc

NULL    =0
DD_OK =0
DDSCL_FULLSCREEN        =       00000001H
DDSCL_EXCLUSIVE         =       00000010H
SM_CXSCREEN         =   0
SM_CYSCREEN         =   1
VK_ESCAPE           = 1BH
DDSD_CAPS           = 00000001H
DDSCAPS_PRIMARYSURFACE      = 00000200H

POINT struc
    x dd ?
    y dd ?
    ends

MSO struc
    hwnd      dd ?
    message  dd ?
    wParam   dd ?
    lParam    dd ?
    time dd ?
    pt          POINT<>
    ends
```

```
WNDCLASS struc
    style           dd ?
    lpfnWndProc     dd ?
    cbClsExtra      dd ?
    cbWndExtra      dd ?
    hInstance dd ?
    hIcon           dd ?
    hCursor         dd ?
    hbrBackground       dd ?
    lpszMenuName        dd ?
    lpszClassName       dd ?
    ends

IDirectDrawVtbl     STRUC
    ; IUnknown methods
    QueryInterfacedd ?
    AddRef   dd ?
    Release   dd ?
    ; IDirectDraw methods
    DD_Compact          dd ?
    DD_CreateClipper  dd ?
    DD_CreatePalette   dd ?
    DD_CreateSurface  dd ?
    DD_DuplicateSurface           dd ?
    DD_EnumDisplayModes          dd ?
    DD_EnumSurfaces dd ?
    DD_FlipToGDISurface           dd ?
    DD_GetCaps                    dd ?
    DD_GetDisplayMode      dd ?
    DD_GetFourCCCodes      dd ?
    DD_GetGDISurface       dd ?
    DD_GetMonitorFrequency      dd ?
    DD_GetScanLine              dd ?
    DD_GetVerticalBlankStatus   dd ?
    DD_Initialize               dd ?
    DD_RestoreDisplayMode       dd ?
    DD_SetCooperativeLevel      dd ?
    DD_SetDisplayMode           dd ?
```

```
    DD_WaitForVerticalBlank     dd ?
    ENDS

IDirectDrawSurfaceVtbl STRUC
    ; IUnknown methods
    QueryInterfacedd ?
    AddRef  dd ?
    Release  dd ?
    ; IDirectDrawSurface methods
    DDS_AddAttachedSurface     dd ?
    DDS_AddOverlayDirtyRect  dd ?
    DDS_Blt dd ?
    DDS_BltBatchdd ?
    DDS_BltFast  dd ?
    DDS_DeleteAttachedSurface dd ?
    DDS_EnumAttachedSurfaces dd ?
    DDS_EnumOverlayZOrders  dd ?
    DDS_Flip                    dd ?
    DDS_GetAttachedSurface      dd ?
    DDS_GetBltStatus            dd ?
    DDS_GetCaps                 dd ?
    DDS_GetClipper              dd ?
    DDS_GetColorKey             dd ?
    DDS_GetDC                   dd ?
    DDS_GetFlipStatus dd ?
    DDS_GetOverlayPosition      dd ?
    DDS_GetPalette              dd ?
    DDS_GetPixelFormat          dd ?
    DDS_GetSurfaceDesc          dd ?
    DDS_Initializedd ?
    DDS_IsLost   dd ?
    DDS_mLock   dd ?
    DDS_ReleaseDC    dd ?
    DDS_Restore         dd ?
    DDS_SetClipper              dd ?
    DDS_SetColorKey             dd ?
    DDS_SetOverlayPosition      dd ?
    DDS_SetPalette      dd ?
```

```
    DDS_Unlock  dd ?
    DDS_UpdateOverlay          dd ?
    DDS_UpdateOverlayDisplay dd ?
    DDS_UpdateOverlayZOrder  dd ?
    ENDS

DDPIXELFORMAT      STRUC
    dwSize                DWORD ?
    dwFlags               DWORD ?
    dwFourCC              DWORD ?
    dwRGBBitCount         DWORD ?
    dwRBitMask            DWORD ?
    dwGBitMask            DWORD ?
    dwBBitMask            DWORD ?
    dwRGBAlphaBitMask     DWORD ?
    ENDS

DDCOLORKEY              STRUC
    dwColorSpaceLowValue      DWORD ?
    dwColorSpaceHighValue     DWORD ?
    ENDS

DDSCAPS         STRUC
    dwCaps              DWORD      ?
    ENDS

DDSURFACEDESC      STRUC
    dwSize                DWORD ?
    dwFlags               DWORD ?
    dwHeightDWORD ?
    dwWidth DWORD ?
    lPitch                DWORD ?
    dwBackBufferCount     DWORD ?
    dwMipMapCount   DWORD ?
    dwAlphaBitDepth   DWORD ?
    dwReserved          DWORD ?
    lpSurface       DWORD ?
    ddckCKDestOverlay      DDCOLORKEY<>
```

```
    ddckCKDestBlt          DDCOLORKEY<>
    ddckCKSrcOverlay       DDCOLORKEY<>
    ddckCKSrcBlt           DDCOLORKEY<>
    ddpfPixelFormat        DDPIXELFORMAT<>
    ddsCaps                DDSCAPS<>
    ENDS

.data
    Text          db 'Left ButtonDown',0
    myclassname   db 'MyClass',0
    caption       db 'Direct Draw Sample',0
    Err           db 'Error',0
    szMsg1        db 'Hello ,DirectDraw !',0
    szMsg2        db 'To exit .Press ESC',0
    align 4
    hInst         dd ?
    hWnd          dd ?
    hdc           dd 0
    lpDD          dd 0
    lpDDSPrimary       dd 0
    msg       MSG<>
    wc        WNDCLASS<>
    ddsd DDSURFACEDESC<>

.code
main:
    call  InitMainWindow    ;初始化主窗口
    call  InitDirectDraw    ;初始化 DirectDraw

MsgLoop:                ;消息循环
    call     GetMessageA,offset msg,0,0,0
    or       eax,eax
    jz       Exit
    call     TranslateMessage,offset msg
    call     DispatchMessageA,offset msg
    jmp      MsgLoop

Exit:
```

```
        call      ExitProcess,0

    ;**********************************************************
    ;消息处理函数
    WinMsgProc      proc   uses   ebx   edi   esi,  handle:DWORD,   wMsg:DWORD,
wParam1:DWORD, lParam1:DWORD
        cmp wMsg,WM_CREATE
        jz    create
        cmp wMsg,WM_DESTROY
        jz    destory
        cmp wMsg,WM_LBUTTONDOWN
        jz    lbuttondown
        cmp wMsg,WM_KEYDOWN
        jz    keydown
        jmp  default

    create:
        xor  eax,eax
        ret

    keydown:
        mov ebx,wParam1
        cmp ebx,VK_ESCAPE         ;按下了 ESC?
        jnz  default
        jmp  destory

    destory:                      ;退出
        call  FreeDirectDraw
        call  PostQuitMessage,0
        xor   eax,eax
        ret

    lbuttondown:
        call  DrawIt
        xor   eax,eax
        ret

    default:
```

```
        call  DefWindowProcA,handle,wMsg,wParam1,lParam1
        ret

WinMsgProc   endp
;*************************************************************
;初始化主窗口
InitMainWindow proc
        call    GetModuleHandleA,0
        mov     [hInst],eax

        mov     wc.lpszClassName,offset myclassname
        mov     eax,[hInst]
        mov     wc.hInstance,eax
        mov     wc.lpfnWndProc,offset WinMsgProc
        call    LoadIconA,0,IDI_APPLICATION
        mov     wc.hIcon,eax
        call    LoadCursorA,0,IDC_IBEAM
        mov     wc.hCursor,eax
        mov     wc.lpszMenuName,0
        call    GetStockObject,WHITE_BRUSH
        mov     wc.hbrBackground,eax
        mov     wc.style,CS_HREDRAW or CS_VREDRAW or CS_GLOBALCLASS
        mov     wc.cbClsExtra,0
        mov     wc.cbWndExtra,0

        call    RegisterClassA,offset wc       ;注册窗口类

        call    GetSystemMetrics,SM_CXSCREEN
        push eax
        call    GetSystemMetrics,SM_CYSCREEN
        pop     ebx
        ;建立窗口
        call    CreateWindowExA, 0, offset myclassname, offset caption, WS_POPUP, 0, 0,
ebx, eax, 0,0,[hInst],0
        mov     [hWnd],eax

        call    ShowWindow,[hWnd],1
        call    UpdateWindow,[hWnd]
```

```
        ret
InitMainWindow endp
;********************************************************************
;初始化 DirectDraw
InitDirectDraw proc
        ;生成 DirectDraw 对象
        call    DirectDrawCreate,NULL,offset lpDD,NULL
        cmp     eax,DD_OK
        jnz     @Error
        ;独占、全屏模式
        mov     eax,lpDD
        mov     eax,[eax]
        call    [eax+IDirectDrawVtbl.DD_SetCooperativeLevel],   lpDD,   hWnd,
DDSCL_EXCLUSIVE OR DDSCL_FULLSCREEN
        cmp     eax,DD_OK
        jnz     @Error
        ;设置显示分辨率
        mov eax,lpDD
        mov eax,[eax]
        call    [eax+IDirectDrawVtbl.DD_SetDisplayMode],lpDD,640,480,8
        cmp     eax,DD_OK
        jnz     @Error
        ;建立主页面
        mov     ddsd.dwSize,size DDSURFACEDESC
        mov     ddsd.dwFlags , DDSD_CAPS
        mov     ddsd.ddsCaps.dwCaps , DDSCAPS_PRIMARYSURFACE
        mov     eax,lpDD
        mov     eax,[eax]
        call    [eax+IDirectDrawVtbl.DD_CreateSurface],lpDD,offset   ddsd,   offset
lpDDSPrimary, NULL
        cmp     eax,DD_OK
        jnz     @Error
        ret
@Error:
        call    Fail
        ret
InitDirectDraw endp
;********************************************************************
```

```
;用 DirectDraw 的兼容 DC 在主窗口上输出文字和图案
DrawIt    proc
     ;取 DC
     mov       eax,lpDDSPrimary
     mov       eax,[eax]
     call      [eax+IDirectDrawSurfaceVtbl.DDS_GetDC],lpDDSPrimary,offset hdc
     ;输出
     call      SetBkColor, hdc,0FF0000H;RGB( 0, 0, 255 )
     call      SetTextColor, hdc, 0FFFFH;RGB( 255, 255, 0 )
     call      TextOutA, hdc, 200, 200,offset szMsg1, 19
     call      TextOutA, hdc, 200, 220,offset szMsg2, 18
     call      Rectangle, hdc, 50,100,200,200
     call      MoveToEx,hdc,0,0,NULL
     call      LineTo,hdc,640,480
     ;输出 DC
     mov       eax,lpDDSPrimary
     mov       eax,[eax]
     call      [eax+IDirectDrawSurfaceVtbl.DDS_ReleaseDC],lpDDSPrimary,hdc
     ret
DrawIt    endp
;*******************************************************
;删除 DirectDraw 和 DirectDrawSuface 对象
FreeDirectDraw proc
     cmp       lpDDSPrimary,NULL
     jz        FreeDD
     mov       eax,lpDDSPrimary
     mov eax,[eax]
     call      [eax+IDirectDrawSurfaceVtbl.Release],lpDDSPrimary
FreeDD:
     cmp       lpDD,NULL
     jz        FreeDirectDrawEnd
     mov       eax,lpDD
     mov       eax,[eax]
     call      [eax+IDirectDrawVtbl.Release],lpDD
FreeDirectDrawEnd:
     ret
FreeDirectDraw endp
;*******************************************************
```

```
Fail        proc
    call        MessageBoxA,NULL,offset Err,offset caption,MB_OK
    call        ExitProcess,0
Fail        endp
end main
```

因为这个程序用到 Direct Draw 的函数，因此，要编译这个程序还要一个 Drirec Draw 的库文件（.lib)。请先确认你已安装了 DirectX，并找出 ddraw.dll 文件，用如下命令生成 ddraw.lib 文件：

```
C:\PWIN98\SYSTEM> implib ddraw.lib ddraw.dll
```

并把 ddraw.lib 复制到程序源文件目录下，在 Makefile 中加入 ddraw.lib：

```
IMPORT=$(MAKEDIR)\..\lib\import32    ddraw.lib
```

8.5 遍历窗口

Windows 提供可以让你遍历系统中所有窗口的函数：

```
MyProc proc , hwin:DWORD , param:DWORD
;做对窗口的处理
MyProc endp
Call    EnumWindows,offset MyProc,MyParam
```

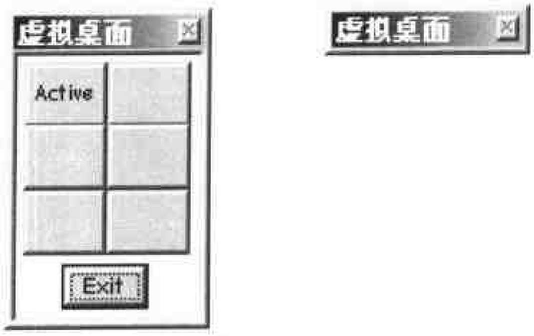
其中：

1）MyParam 是传给 MyProc 的参数 param。

2）MyProc 是自定义的函数，它接收所遍历到的窗口的句柄和调用 EnumWindows 时传给它的参数。当希望处理下一个窗口时，函数应返回非 0，否则返回 0。

3）EnumWindows 只遍历顶层窗口，要遍历指定窗口的子窗口，可以用 EnumChildWindows 函数，它的用法与 EnumWindows 大致相同。

当打开了太多的窗口时，可能要为找不到所需的窗口而烦恼。下面的例子或许可以解决此问题。如图 8-5 所示，这个程序将打开的每一个窗口都模拟为一个桌面，用户可以在不同的桌面间来回切换，在不同的桌面开不同的窗口。



a)平常状态　　b)最小化时

图 8-5

程序界面上有 6 个代表桌面的按钮，当用户按下其中一个时，程序遍历系统所有窗口，并查看它的风格，若有 WS_VISIBLE 风格，说明它是可见的。于是，将它的窗口句柄保存到当前“桌面”对应的内存中，并用 ShowWindow API 把这个窗口设为不可视。然后，把要转向的“桌面”中所保存的窗口句柄，用 ShowWindow API 将对应的窗口设为可视。

这个遍历过程中有 3 个遇到的窗口句柄是不应隐藏的。一个当然是程序自身的主界面；另一个是任务栏，它的类名是“Shell_TrayWnd”；还有一个是程序管理器，它的标题是“Program Manager”。因此，在程序开始时，我们先遍历所有窗口，用 GetClassNameA 和 GetWindowTextA 找出这两个程序的窗口句柄，并记下它，以后隐藏窗口时先与这 3 个窗口句柄对比，若相等则不隐藏。

```
;***************************
;文件：Pager.asm           *
;功能：虚拟 6 个桌面       *
;***************************
.386p
.model flat,stdcall
include win32.inc
extrn MessageBoxA:proc
extrn ExitProcess:proc
extrn GetModuleHandleA:proc
extrn DialogBoxParamA:proc
extrn EndDialog:Proc
extrn GetWindowRect:proc
extrn MoveWindow:proc
extrn GetWindowLongA:proc
extrn SetDlgItemTextA:proc
extrn ShowWindow:proc
extrn GetWindowTextA:proc
extrn GetClassNameA:proc
extrn lstrcmpA:proc
extrn EnumWindows:proc

IDC_BUTTON0     =100
IDC_BUTTON1     =101
IDC_BUTTON2     =102
IDC_BUTTON3     =103
IDC_BUTTON4     =104
IDC_BUTTON5     =105
```

```
TRUE   = 1
FALSE = 0
GWL_STYLE            = (-16)
RECT      struc
        left            dd ?
        top             dd ?
        right           dd ?
        bottom          dd ?
RECT      ends

.data
    DialogName   db 'MyDialog',0
    CaptionFail  db '失败！',0
    TextFail  db 'API:DialogBoxParamA 调用失败！',0

    MAXWINCOUNT =  150
    Handles       dd 6*MAXWINCOUNT dup (?)
    WinCount      dd 6 dup (0)

    hMainWin      dd ?
    TitleTmp      db 256 dup (?)
    TitleManager  db 'Program Manager',0
    ClassShell    db 'Shell_TrayWnd',0
    hManager      dd ?
    hShell        dd ?
    now           dd 0
    OffsetHandle  dd ?

    min           dd 0
    rect          RECT<>
    NotActive     db 0
    Active        db 'Active',0

.code
Start:
    call  GetModuleHandleA,0          ;取得模块句柄，返回结果在 eax 中
    ;调用 DialogBoxParamA 显示对话框
    call  DialogBoxParamA,eax,offset DialogName,0,offset DialogProc,0
```

```
        cmp eax,0                   ;成功否?
        jnz  Exit                   ;成功则退出
        ;显示出错信息
        call  MessageBoxA,0,offset TextFail,offset CaptionFail,MB_OK
    Exit:
        call  ExitProcess,0         ;退出进程
    ;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    ;对话框消息处理过程
    DialogProc   proc uses ebx edi esi, hWnd:DWORD, wMsg:DWORD, wParam:DWORD, lParam:DWORD
        cmp wMsg,WM_INITDIALOG
        jz        WmInitDialog
        cmp       wMsg,WM_CLOSE
        jz        WmClose
        cmp       wMsg,WM_NCLBUTTONDBLCLK
        jz        Lessen
        cmp       wMsg,WM_COMMAND
        jz        WmCommand
        jmp       ExitProc
    WmInitDialog:                   ;对话框初始化处理
        call      InitDlg,hWnd
        jmp       ExitProc
    WmClose:                        ;用户按下了 Alt+F4 或单击右上角 X 按钮
        call      ShowAll
        call      EndDialog,hWnd,1      ;关闭对话框
        Jmp       ExitProc
    Lessen:
        call      GetWindowRect,hWnd,offset rect
        mov       ebx,rect.right
        sub       ebx,rect.left
        mov eax,[min]
        or        eax,eax
        jnz       max
        call      MoveWindow,hWnd,rect.left,rect.top,ebx,22,TRUE
        mov       [min],TRUE
        jmp       ExitProc
    max:
        call      MoveWindow,hWnd,rect.left,rect.top,ebx,143,TRUE
```

```
        mov     [min],FALSE
        jmp     ExitProc
WmCommand:
        mov     ebx,wParam
        cmp     bx,IDOK             ;是否按下了“退出”按钮?
        jz      OKPress
        cmp     bx,IDC_BUTTON0
        jz      Button0Press
        cmp     bx,IDC_BUTTON1
        jz      Button1Press
        cmp     bx,IDC_BUTTON2
        jz      Button2Press
        cmp     bx,IDC_BUTTON3
        jz      Button3Press
        cmp     bx,IDC_BUTTON4
        jz      Button4Press
        cmp     bx,IDC_BUTTON5
        jz      Button5Press
OKPress:
        jmp     WmClose             ;关闭对话框

Button0Press:                   ;用户点击了桌面 0
        call    ChangeTo,0
        jmp     ExitProc
Button1Press:                   ;用户点击了桌面 1
        call    ChangeTo,1
        jmp     ExitProc
Button2Press:                   ;用户点击了桌面 2
        call    ChangeTo,2
        jmp     ExitProc
Button3Press:                   ;用户点击了桌面 3
        call    ChangeTo,3
        jmp     ExitProc
Button4Press:                   ;用户点击了桌面 4
        call    ChangeTo,4
        jmp     ExitProc
Button5Press:                   ;用户点击了桌面 5
        call    ChangeTo,5
```

```
        jmp     ExitProc

ExitProc:                       ;退出消息处理
        xor     eax,eax
        ret
DialogProc endp

;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;取 Program Manager 和任务栏句柄
GetSysWin proc , hwin:DWORD , param:DWORD
        call    GetClassNameA,hwin,offset TitleTmp,255
        call    lstrcmpA,offset TitleTmp,offset ClassShell
        or      eax,eax
        jnz     Next
        mov     eax,hwin
        mov     [hShell],eax
        jmp     GetSysWinExit
Next:
        call    GetWindowTextA,hwin,offset TitleTmp,255
        call    lstrcmpA,offset TitleTmp,offset TitleManager
        or      eax,eax
        jnz     GetSysWinExit
        mov     eax,hwin
        mov     [hManager],eax
GetSysWinExit:
        mov     eax,1
        ret
GetSysWin endp
;初始化
InitDlg proc, hdlg1:DWORD
        mov     eax,hdlg1
        mov     [hMainWin],eax
        call    SetDlgItemTextA,eax,100,offset Active

        call    EnumWindows,offset GetSysWin,0
        ret
InitDlg endp
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
;保存现在的桌面的可视窗口
HideNow proc , hwin1:DWORD , param1:DWORD
        mov     eax,[hMainWin]
        cmp     hwin1,eax
        jz      HideNowExit
        mov     eax,[hShell]
        cmp     hwin1,eax
        jz      HideNowExit
        mov     eax,[hManager]
        cmp     hwin1,eax
        jz      HideNowExit
        call    GetWindowLongA,hwin1,GWL_STYLE
        test    eax,WS_VISIBLE
        jz      HideNowExit

        mov     ecx,[OffsetHandle]
        mov     eax,hwin1
        mov     [ecx],eax
        add     ecx,4
        mov     [OffsetHandle],ecx

        mov     eax,[now]
        shl     eax,2
        inc     dword ptr [WinCount+eax]
        call    ShowWindow,hwin1,SW_HIDE
HideNowExit:
        mov     eax,1
        ret
HideNow endp

;转到用户指定的桌面
ChangeTo proc ,to:DWORD
        mov     eax,[now]
        add     eax,100
        call    SetDlgItemTextA,[hMainWin],eax,offset NotActive
        mov     eax,to
        add     eax,100
        call    SetDlgItemTextA,[hMainWin],eax,offset Active
```

```
        mov     eax,to
        cmp     eax,[now]
        jz      ChangeToExit

        mov     eax,[now]
        mov     ebx, MAXWINCOUNT*4
        mul     ebx
        mov     esi,offset Handles
        add     eax,esi
        mov     [OffsetHandle],eax

        mov     eax,[now]
        mov     dword ptr [WinCount+eax*4],0
        call    EnumWindows,offset HideNow,0
        mov     eax,to
        mov     [now],eax

        mov     ecx,[WinCount+eax*4]
        mov     ebx,MAXWINCOUNT*4
        mul     ebx
        mov     esi,offset Handles
        add     esi,eax
again:
        jecxz   ChangeToExit
        push    ecx
        call    ShowWindow,dword ptr [esi],SW_SHOW
        pop     ecx
        dec     ecx
        add     esi,4
        jmp     again

ChangeToExit:
        ret
ChangeTo endp
;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;程序退出时显示所有保存的窗口
ShowAll proc
        mov     edx,offset WinCount
```

```
        mov     esi,offset Handles

 show1:
        mov     ecx,[edx]
        jcxz    ok

        push    esi
 show2:
        push    esi
        push    ecx
        push    edx
        call    ShowWindow,dword ptr [esi],SW_SHOW
        pop     edx
        pop     ecx
        pop     esi
        add     esi,4
        dec     ecx
        jnz     show2
        pop     esi
ok:
        add     edx,4
        add     esi,MAXWINCOUNT*4
        cmp     edx,offset WinCount + 4*6
        jb      show1

        ret
ShowAll endp
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
end Start
```

第九章　PE 文件格式

本章主要介绍 Win 32 可移植可执行文件（PE）的详细格式。包括分析 PE 文件结构：DOS 插桩程序、NT 映像头、节表和节；各种 PE 文件格式查看器等内容。

9.1 PE 文件结构

Windows 9X/NT 的 PE（Portable Executable 可移植可执行）文件格式是 Win32 可执行文件的标准格式。了解其格式对理解 Win32 进程的装载、内存管理、系统的原理有很大的帮助。

在讨论 PE 文件结构之前，先要理解一个概念：相对虚地址（Relative Virtual Address，简称 RVA）。它是一个相对于可执行文件映射到内存的基地址偏移量。

例如，当可执行文件映射到内存时的基地址是 400000H，则 RVA 地址 12345H 的实际内存地址是 412345H。

PE 文件结构如图 9-1 所示。

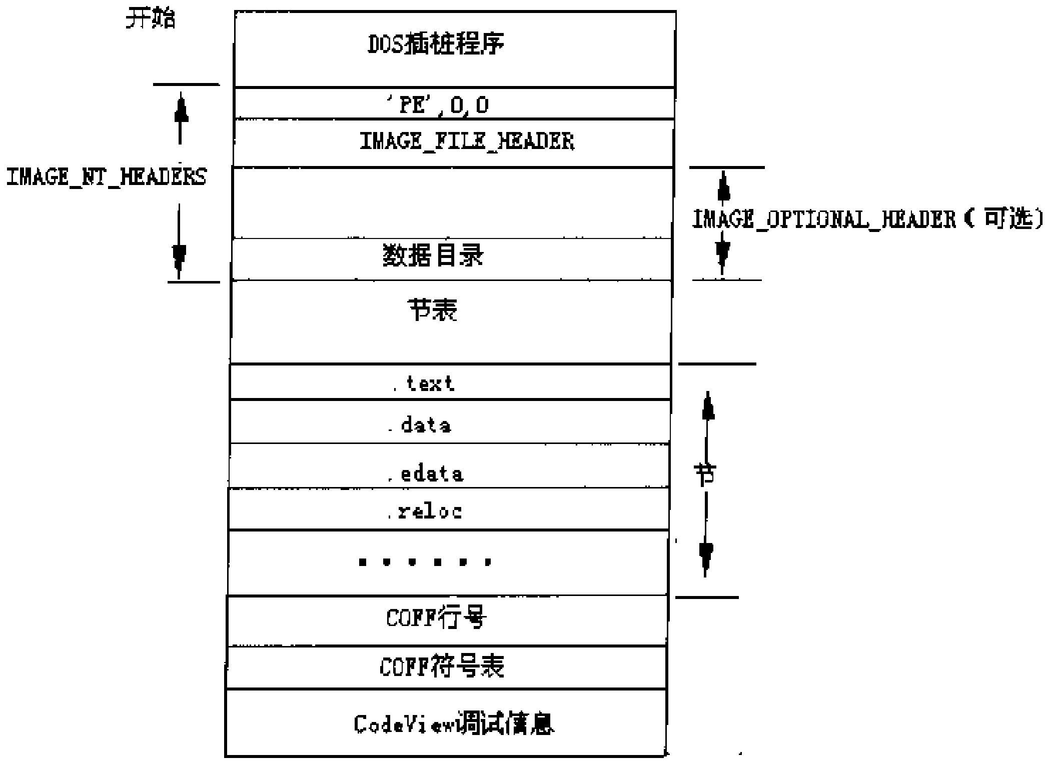


图 9-1

9.1.1 DOS 插桩程序

在开始处的 DOS 小程序是一个显示"This program can not be run in DOS mode"或"This

program must be run under Win32"之类信息的小程序，一般可通过 DOS 功能调用 Int 21H 的 9 号功能（ah=9）来显示。如：

```
C:\PWIN98>debug cdplayer.exe

-u

0000 0E          PUSH   CS
0001 1F          POP    DS
0002 BA0E00      MOV    DX,000E
0005 B409        MOV    AH,09
0007 CD21        INT    21
0009 B8014C      MOV    AX,4C01
000C CD21        INT    21

-d cs:0e

0000                                               54 68      Th
0010  69 73 20 70 72 6F 67 72-61 6D 20 63 61 6E 6E 6F   is program canno
0020  74 20 62 65 20 72 75 6E-20 69 6E 20 44 4F 53 20   t be run in DOS
0030  6D 6F 64 65 2E 0D 0D 0A-24 00 00 00 00 00 00 00   mode....$.......
```

如在 DOS 下试图运行这个 Windows 程序时，将显示该错误信息。这个小程序一般是编译器加上去的默认值，可用其它程序替代它。

如果用户希望 Windows 程序在 DOS 下运行时有更详细的错误提示信息，或程序可以同时在 DOS 和 Windows 下正常运行时，这种方法是有用的。

上述情况下的一个典型例子是 Windows 9x 下的程序 Winver.exe。这个程序在 Windows 下运行时将出现如图 9-2 所示的对话框，以说明当前使用的 Windows 版本。

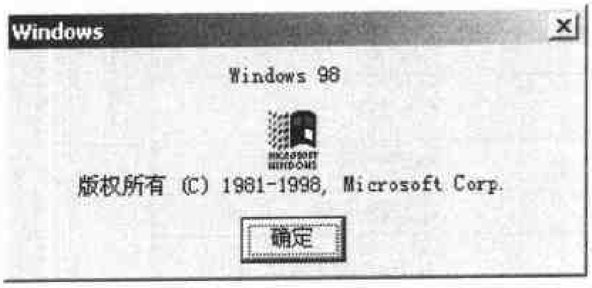


图 9-2

但在 DOS 下运行时，将显示下面的信息。

```
C:\PWIN98>winver.exe
Windows 98
```

下面的 Windows 程序在 Windows 下运行时显示如图 9-3 所示的信息。现在我们希望

这个 Windows 程序在 DOS 下运行时有一个不同的提示。

图 9-3

```
;*****************************
;Stub.asm                    *
;功能：演示改变 EXE 文件的     *
;          DOS 插桩程序       *
;*****************************
NULL   = 0               ;常量定义
MB_OK = 0                ;

.386p
.model flat                  ;平坦内存模式
extrn MessageBoxA:proc       ;外部过程声明
extrn ExitProcess:proc       ;

.data
    Caption db 'OK',0
    Text    db '这是在 Windows 下运行的情况',0

.code
main:
    push MB_OK             ;MessageBoxA 的参数入栈
    push offset Caption    ;
    push offset Text       ;
    push NULL              ;
    call MessageBoxA
    push 0                 ;ExtiProcess 的参数入栈
    call ExitProcess

end main
```

首先，我们要有自己的 DOS 程序：

```
;********************************
;文件：dos.asm                  *
```

```
;功能：作为 Windows 可执行文件的  *
;      插桩程序，显示一个字符串   *
;*********************************

data segment
    MSG     db 'This is the DOS STUB speaking.',13,10,'$'

data ends
code segment
    assume cs:code,ds:data

start:
    mov ax,data
    mov ds,ax
    mov dx,offset MSG
    mov ah,9
    int 21H
    mov ah,4cH
    int 21H

code ends
    end start
```

这个 DOS 程序在屏幕上显示一个字符串："This is the DOS STUB speaking."，这就是我们的 Windows 程序在 DOS 下运行的结果。这个 DOS 程序可以随意，可以很大，也可以很小，可用别的 DOS 程序代替它。

下面将是关键的步骤，建立一个定义文件 stub.def 为：

```
NAME            stub.def
DESCRIPTION 'STUB EXAMPLE'
STUB            'dos.exe'
CODE            PRELOAD MOVEABLE DISCARDABLE
DATA            PRELOAD MOVEABLE MULTIPLE

EXETYPE         WINDOWS
```

最重要的一句是第 4 行的"STUB 'dos.exe'"。

它定义了插桩程序使用的 DOS 可执行文件。Stub.ASM 的 Makefile 如下面的代码。它主要是在连接时增加了 DEF 文件，使 tlink32 知道插桩程序的名字。

```
NAME = STUB
OBJS = $(NAME).obj
```

```
DEF = $(NAME).def
IMPORT=$(MAKEDIR)\..\lib\import32.lib
$(NAME).EXE: $(OBJS) $(DEF)
  tlink32 /Tpe /aa /c   $(OBJS),$(NAME),, $(IMPORT), $(DEF)
.asm.obj:
   tasm32 /ml $&.asm
```

程序在 Windows 中运行的结果如图 9-3 所示，在 DOS 下运行将显示如下信息：

```
F:\Bk\Chapter8\stub>stub
This is the DOS STUB speaking.
```

如果你希望在 Windows 的 DOS 窗口运行这个程序，而又有纯 DOS 下运行的效果，可用 debug 调入该程序并执行：

```
F:\Bk\Chapter9\stub>DEBUG STUB.EXE
-G
This is the DOS STUB speaking.
Program terminated normally
```

9.1.2 NT 映像头

紧跟 DOS 小程序后面的便是 PE 文件的 NT 映像头（IMAGE_NT_HEADERS），它存放 PE 整个文件信息分布的重要信息。

NT 映像头由三部分组成。

1. 字串 "PE\0\0"

这个"PE"字符串是一个标志，它是 NT 映像头的开始，也是 PE 文件中与 Windows 有关的内容的开始。这个字串在文件中的位置，可以在 DOS 程序头中找到它的指针，它占用 4 个字节，位于文件开始偏移 3CH 字节中。如下是 Hello.exe 的例子，偏移 3CH 中的粗体是字串"PE"的偏移，低位在前，高位在后，即 100H。

```
0000  4D 5A 50 00 02 00 00 00-04 00 0F 00 FF FF 00 00   MZP.............
0010  B8 00 00 00 00 00 00 00-40 00 1A 00 00 00 00 00   ........@.......
0020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0030  00 00 00 00 00 00 00 00-00 00 00 00 00 01 00 00   ................
0040  BA 10 00 0E 1F B4 09 CD-21 B8 01 4C CD 21 90 90   ........!..L.!..
0050  54 68 69 73 20 70 72 6F-67 72 61 6D 20 6D 75 73   This program mus
......
0100  50 45 00 00 4C 01 04 00-75 27 7B B2 00 00 00 00   PE..L...u'{.....
0110  00 00 00 00 E0 00 8E 81-0B 01 02 19 00 02 00 00   ................
......
```

2. 映像文件头

紧跟着"PE\0\0"的是映像文件头（IMAGE_FILE_HEADER）。映像文件头是 NT 映

像头的主要部分，它包含有PE文件的最基本的信息。

它的由几项组成，如表9-1所示。

表9-1 映像文件头的基本信息

顺序	名字	大小（字节）	描述
1	Machine	2	机器类型，X86值为14CH
2	NumberOfSection	2	文件中节的个数
3	TimeDataStamp	4	生成该文件的时间
4	PointerToSymbolTable	4	COFF符号表的偏移
5	NumberOfSymbols	4	符号数目
6	SizeOfOptionalHeader	2	可选头的大小
7	Characteristics	2	标记

下面是Hello.exe的例子，在每一行中都用粗体/正常体间隔来分开各项，这样更容易看一些，本章中其它例子亦是如此：

0100 50 45 00 00 **4C 01** 04 00-**75 27 7B B2** 00 00 00 00 PE..L...u'{.....

(1) (2) (3) (4)

0110 00 00 00 00 **E0 00** 8E 81

(5) (6) (7)

现在我们来详细解释这些内容：

1）机器字（Machine）说明可执行文件需要的CPU类型。PE文件被定义为“可移值”的，它可以在多种机器上使用。

事实上，Windows NT不但可以运行在X86机器上，还可以运行在MIPS、Power PC等多种机器上，如表9-2所示是几种典型的机器的标志。

表9-2 典型的机器类型标志

机器	标志
Intel i386	14CH
MIPS R3000	162H
MIPS R4000	166H
Alpha AXP	184H
Power PC	1F0H

2）文件中节的个数，所谓“节”，是文件中存放一类数据的地方，如数据、代码、重定位表等。关于“节”后面有叙述。

3）文件生成时间。

4）COFF符号表偏移，这是程序的调试信息。PE格式支持多种调试格式。

5）符号数目。即上一字段中所放符号表的数目。

6）可选映象头（IMAGE_OPTIONAL_HEADER）的大小，即NT映象头的第三部分的大小。

7）标记，这是由几个值有选择地通过或运算而得。这些值如表 9-3 所示。

表 9-3 标记值

值	意义
1H	重定位信息被移去
2H	文件可执行（例如，没有不可解决的外部引用）
4H	行号被移去
8H	符号被移去
80H	机器的字节顺序是倒转的（低位）
100H	32 位机器
200H	.DBG 文件的调试信息被移去
400H	如果映象文件是在可移动媒体中，则先复制到交换文件再运行
800H	如果映象文件是在网络中，则复制到交换文件才运行
1000H	系统文件
2000H	文件是 DLL 文件
4000H	文件只能运行在单处理器的机器上
8000H	机器的字节顺序是倒转的（高位）

3. 可选映象头

映象文件头后是一个可选映象头（IMAGE_OPTIONAL_HEADER）类型结构，这是一个可选的结构。

它的各项如表 9-4 所示。

表 9-4 IMAGE_OPTIONAL_HEADER 类型结构

顺序	名字	大小（字节）	描述
1	Magic	2	一般是 010BH
2	MajorLinkerVersion	1	连接程序主版本号
3	MinorLinkerVersion	1	连接程序次版本号
4	SizeOfCode	4	代码的总尺寸
5	SizeOfInitializedData	4	已初始化的数据总尺寸
6	SizeOfUninitializedData	4	未初始化的数据总尺寸
7	AddressOfEntryPoint	4	开始执行位置
8	BaseOfCode	4	代码节开始的位置
9	BaseOfData	4	数据节开始的位置
10	ImageBase	4	可执行文件的默认装入的内存地址
11	SectionAlignment	4	可执行文件装入内存时节的对齐数字

续表 9-4

顺序	名字	大小（字节）	描述
12	FileAlignment	4	文件中节的对齐数字，一般是一个扇区长（512 字节）
13	MajorOperationSystemVersion	2	要求最低操作系统版本号的主版本号
14	MinorOperationSystemVersion	2	要求最低操作系统版本号的次版本号
15	MajorImageVersion	2	可执行文件主版本号
16	MinorImageVersion	2	可执行文件次版本号
17	MajorSubsystemVersion	2	要求最小子系统主版本号
18	MinorSubsystemVersion	2	要求最小子系统次版本号
19	Reserved1	4	保留，一般为 0
20	SizeOfImage	4	装入内存后映象的总尺寸
21	SizeOfHeaders	4	NT 映象头+节表的大小
22	CheckSum	4	检验和
23	Subsystem	2	可执行文件的子系统。如 GUI 子系统、CUI 子系统
24	DllCharacteristics	2	何时 DllMain 被调用，一般为 0
25	SizeOfStackReserve	4	初始化线程时保留的堆栈大小
26	SizeOfStackCommit	4	初始化线程时提交的堆栈大小
27	SizeOfHeapReserve	4	进程初始化时保留的堆大小
28	SizeOfHeapCommit	4	进程初始化时提交的堆大小
29	LoaderFlags	4	此项与调试有关
30	NumberOfRvaAndSize	4	数据目录的项数，一般是 16
31	DataDirectory[]	128	数据目录

Hello.exe 的可选映像头如下所示：

```
0110        0B 01 02 19 00 02 00 00                   ........
             (1) (2) (3)   (4)
0120  00 06 00 00 00 00 00 00-00 10 00 00 00 10 00 00   ................
         (5)         (6)         (7)         (8)
0130  00 20 00 00 00 00 40 00-00 10 00 00 00 02 00 00   .....@.........
         (9)         (10)        (11)        (12)
```

```
0140  01 00 00 00 00 00 00 00-03 00 0A 00 00 00 00 00   ................
          (13)  (14)  (15)  (16)  (17)  (18)   (19)
0150  00 50 00 00 00 04 00 00-00 00 00 00 02 00 00 00   .P..............
          (20)       (21)        (22)  (23) (24)
0160  00 00 10 00 00 20 00 00-00 00 10 00 00 10 00 00   ................
          (25)       (26)        (27)       (28)
0170  00 00 00 00 10 00 00 00-00 00 00 00 00 00 00 00   ................
          (29)       (30)           (31)
0180  00 30 00 00 90 00 00 00-00 00 00 00 00 00 00 00   .0..............
0190  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
01A0  00 40 00 00 14 00 00 00-00 00 00 00 00 00 00 00   .@..............
01B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
01C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
01D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
01E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
01F0  00 00 00 00 00 00 00 00                           ........
```

现在我们来详细解释这些内容：

1）Magic 幻数，这是一个标志，说明这个文件是一个 ROM 映像还是一个普通可执行的映像。这个值一般是 10BH。

2）MajorLinkerVersion，连接程序的主版本号，在这里是 02。

3）MinorLinkerVersion，连接程序的次版本号，在这里是 19H

4）SizeOfCode，代码的总尺寸。这是所有代码加起来的尺寸，并且这个值是向上对齐某一个值的整数倍。例如，在此是 200H，即对齐的是一个磁盘扇区字节数（200H）的整数倍。这个值一般就是.text 节的尺寸。

5）SizeOfInitializeData，已经初始化了的数据尺寸，即在编译时就确定其值的数据的尺寸，对齐的情况同上。但这个数据并不太准确。

6）SizeOfUninitializeData，未初始化的数据尺寸，这是程序装入内存时要为未初始化的数据分配内存的尺寸，这部分数据并不占用文件的尺寸。这个数据一般在.bbs 节中。

7）AddressOfEntryPoint，程序开始执行的地方。这是一个 RVA，这个地址通常指向.text 中的一个位置。

8）BaseOfCode，代码节开始处的 RVA，这个值一般是 1000H。代码一般在数据之前装入内存。

9）BaseOfData，数据节开始处的 RVA。

10）ImageBase，这是可执行文件的默认装入基地址。如果程序装入时是用这个值做基地址，则装入时就不需重定位。对于 EXE 文件，这个值一般是 400000H，但也有可能是 10000H 等。

11）SectionAlignment，内存中节的对齐值。当文件装入内存时，每个节的起始地址都是这个值的整数位。这个值一般是 10000H（4K）或 100000H（64K）。

12）FileAlignment，文件中节的对齐值。文件中每个节都起始于这个值的整数倍处。这个值一般是 200H（512），即磁盘扇区的大小。

13）MajorOperationSystemVersion，要求操作系统的最低版本号的主版本号。这个域与下一个域合在一起，一般是 1.0。

14）MinorOperationSystemVersion，要求系统的最低版本号的次版本号。见上一项。

15）MajorImageVersion，该可执行文件的主版本号，由程序员定义。

16）MinorImageVersion，该可执行文件的次版本号，见上一项。

17）MajorSubsystemVersion，要求的最低子系统版本的主版本号。这个值与下一个域一起，一般是 4.0，即要求 Windows 4.0（Windows 95），但在我们这个例子中是 3.10，即 Windows NT 3.1。

18）MinorSubsystemVersion，要求的最低子系统版本的次版本号。见上一项。

19）Reserved1，保留，一般置 0。

20）SizeOfImage，映像装入内存后的总尺寸，这是一个对齐到 SectionAlignment 的整数倍的尺寸。

21）SizeOfHeaders，头尺寸，这是 NT 映像头与节表的大小的和。

22）CheckSum，这是一个 CRC 检验和。一般的 EXE 文件可以是 0，但是，一些重要的系统 DLL 文件，它必须有一个检验和。

23）System，程序使用的用户接口子系统，如表 9-5 所示是它的列表。

表 9-5 子系统

值	子系统
0	未知
1	不需要子系统
2	图形接口（GUI）子系统
3	字符子系统（CUI）
5	OS/2 字符子系统
7	POSIX 字符子系统
8	保留

24）DllCharacteristics，表示 DllMain()函数何时被调用。这个值一般是 0。

25）SizeOfStackReserve，为线程保留的堆栈大小。这是线程可以使用的最大堆栈尺寸。但是，它并不是一开始就全部提交，它只提交其中一部分，只有当有必要时，才提交剩下的部分。

26）SizeOfStackCommit，当线程初始化时提交的堆栈大小，见上一个域 SizeOfStackReserve。这个值总是比 SizeOfStackReserve 小。

27）SizeOfHeapReserve，为进程缺省堆保留的内存。但是，并不全部提交。这个堆的句柄可以用 Windows API GetProcessHeap()来取得。

28）SizeOfHeapCommit，进程初始化时提交的缺省堆的大小。见上一个域。

29）LoaderFlags，这个项与调试有关，一般为 0。

30）NumberOfRvaAndSize，数据目录（见下一个域）的项数。这个值通常都是 16。

31）DataDirectory[]数据目录，它是一个 IMAGE_DATA_DIRECTORY 数组，里面放的是这个可执行文件的一些重要部分的起始 RVA 和尺寸，目的是使可执行文件的装入更快。数组的项数见上一个域。IMAGE_DATA_DIRECTORY 包含有 2 个域，例如：

```
IMAGE_DATA_DIRECTORY   STRUC
VirtualAddressDD ?
Size      DD ?
IMAGE_DATA_DIRECTORY   ENDS
```

9.1.3 节表

NT 映像头之后是节表（Section table），它是一个 IMAGE_SECTION_HEADER 数组，数组的项数就是文件中的节的个数。IMAGE_SECTION_HEADER 中放的是一个节的信息，如名字、地址、长度、属性等，如表 9-6 所示。

表 9-6　IMAGE_SECTION_HEADER

顺序	名字	大小（字节）	描述
1	Name	8	节名
2	PhyscialAddress 或 VirtualSize	4	OBJ 文件用作表示本节物理地址 EXE 文件中表示节的虚拟尺寸
3	Virtual	4	这个值+映象基地址=本节在内存中的真正地址 在 OBJ 中本节无意义
4	SizeOfRawData	4	本节的原始尺寸
5	PointerToRawData	4	本节原始数据在文件中的位置
6	PointerToRelocations	4	OBJ 中表示该节重定位信息的偏移 EXE 中无意义
7	PointerToLinenumbers	4	行号偏移
8	NumberOfRelocations	2	本节要重定位的数目
9	NumberOfLinenumbers	2	本节在行号表中的行号数目
10	Characteristics	4	节属性

如下所示，Hello.exe 的节表包含有对 4 个节的描述 CODE、DATA、.idata 和.reloc。每个表对应一个 IMAGE_SECTION_HEADER 结构：

```
01F0    43 4F 44 45 00 00 00 00                 CODE....
                 (1)
0200  00 10 00 00 00 10 00 00-00 02 00 00 00 06 00 00   ................
         (2)         (3)         (4)         (5)
0210  00 00 00 00 00 00 00 00-00 00 00 00 20 00 00 60   ...........`
         (6)         (7)        (8)    (9)      (10)
0220  44 41 54 41 00 00 00 00-00 10 00 00 00 20 00 00   DATA..........
            (1)                (2)          (3)
```

```
0230  00 02 00 00 00 08 00 00-00 00 00 00 00 00 00 00   ................
          (4)       (5)          (6)         (7)
0240  00 00 00 00 40 00 00 C0-2E 69 64 61 74 61 00 00   ....@....idata..
      (8) (9)  (10)           (1)
0250  00 10 00 00 00 30 00 00-00 02 00 00 00 0A 00 00   .....0..........
      (2)    (3)       (4)          (5)
0260  00 00 00 00 00 00 00 00-00 00 00 00 40 00 00 C0   ............@...
      (6)       (7)    (8)   (9)   (10)
0270  2E 72 65 6C 6F 63 00 00-00 10 00 00 00 40 00 00   .reloc.......@..
         (1)         (2)       (3)
0280  00 02 00 00 00 0C 00 00-00 00 00 00 00 00 00 00   ................
          (4)       (5)           (6)         (7)
0290  00 00 00 00 40 00 00 50                           ....@..P
        (8)  (9)     (10)
```

解释：

1）Name，是该节的名字，这个名字最长是 8 个字节。一般常见的名字有.text、CODE、.data、DATA、.idata、.reloc 等。

2）PhyscialAddress 或 VirtualSize。在 OBJ 文件中，这个域表示节的物理地址（PhyscialAddress），在 EXE 文件中，这个域是代码或数据节的虚拟尺寸（VirtualSize）。有时，这个域和第 4 个域 SizeOfRawData 换转。

3）VirtualAddress，这个域在 OBJ 文件中没有意义；在 EXE 文件中，这是一个 RVA，表明本节在装入内存时的地址。即本地址+基地址=本节在内存中的真正地址。

4）SizeOfData，本节的原始尺寸。在 EXE 文件中，这是对齐到文件对齐尺寸的整数倍后的值，在 OBJ 文件中，这是一个精确的值（即未作对齐处理）。

5）PointerToRawData，本节原始数据在文件中的位置。

6）PointerToRelocations，这个域在 EXE 文件中无意义；在 OBJ 文件中，它表示本节的重定位信息在文件的偏移值。

7）PointerToLinenumbers，行号表在文件中的偏移值。这是文件的调试信息。

8）NumberOfRelocations，在 EXE 文件中本域无意义；在 OBJ 文件中，这是本节在重定位表中的重定位数目。

9）NumberOfLinenumbers，本节在行号表中的行号数目。

10）节的属性，其意义如表 9-7 所示。

表 9-7　节属性

值	意义
8	保留
20H	包含代码
40H	包含已初始化的数据
80H	包含未初始化的数据

续表 9-7

值	意义
100H	连接器使用，保留
200H	连接器使用，保存有注释或其它连接器使用的数据
800H	连接器使用
1000H	连接器使用
100000H	1 字节对齐
200000H	2 字节对齐
300000H	4 字节对齐
400000H	8 字节对齐
500000H	16 字节对齐，这是默认值
600000H	32 字节对齐
700000H	64 字节对齐
1000000H	包含扩展的重定位数据?????
2000000H	节可以被丢弃
4000000H	不使用 cache 的
8000000H	不分页的
10000000H	共享的
20000000H	可执行的
40000000H	可读的
80000000H	可写的

9.1.4 节

"节（Section）"跟在节表之后，一般 PE 文件都会有几个"节"。PE 文件有很多种"节"，我们这里要讲几种比较常见"节"。

1. 代码节

代码节一般名为.text 或 CODE，该节含有程序的可执行代码。每个 PE 文件都会有代码节。在代码节中，还有一些特别的数据，是作为调用引入函数之用，例如，对 API GetMessageBoxA 调用：

```
push MB_OK
push offset Caption
push offset Text
push NULL
call MessageBoxA
```

其反汇编如下：

```
:00401000 6A00              push 00000000
:00401002 6800204000        push 00402000
:00401007 680D204000        push 0040200D
:0040100C 6A00              push 00000000
```

```
:0040100E E807000000          Call 0040101A
......
:0040101A FF254C304000        Jmp dword ptr [0040304C]
......
```

指令 call MessageBoxA 被换成了指令 call 0040101A，地址 40101AH 仍在.text 节中，它放有一个跳转指令：Jmp dword ptr[0040304C]，即这个跳转指令的目的地址放在 0040304CH 中；地址 0040304CH 是处于.idata 节中，其中放的才是 MessageBoxA 的真正地址，如图 9-4 所示。

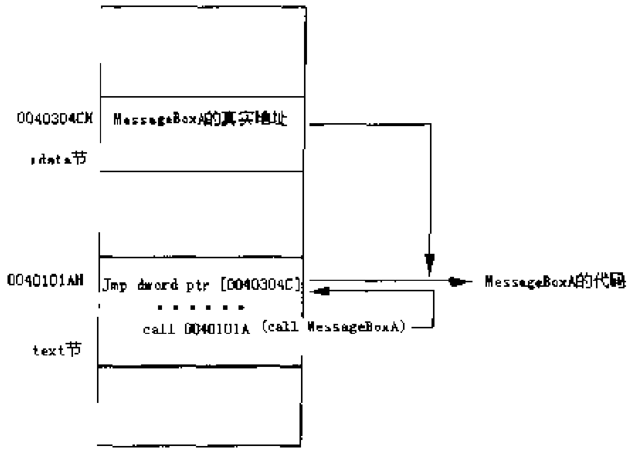


图 9-4

上面是通常的情况，但编译器也有可能不产生 jmp 指令，而直接产生一个指令 call dword ptr [004030c]，这省却了一条跳转指令。

2. 已初始化的数据节

已初始化的数据节中放的是在编译时刻就已确定的数据。例如，在我们的 Hello World 程序中，两个字符串“Hello World!”和“欢迎来到 Win32 汇编的世界”就是已初始化数据的一个例子。这个节一般取名是.data，但有时也取名叫 DATA。

```
0800   48 65 6C 6C 6F 20 57 6F-72 6C 64 21 00 BB B6 D3    Hello World!....
0810   AD C0 B4 B5 BD 57 69 6E-33 32 BB E3 B1 E0 B5 C4    .....Win32......
0820   CA C0 BD E7 00 00 00 00-00 00 00 00 00 00 00 00    ................
```

3. 未初始化数据节

这个节里放有未初始化的全局变量和静态变量。节的名称一般叫.bbs。我们所使用的 Tlink32 并不产生.bbs 节，而扩展 DATA 节来代替。

```
0600   6A 00 68 00 20 40 00 68-0D 20 40 00 6A 00 E8 07    j.h. @.h. @.j...
0610   00 00 00 6A 00 E8 06 00-00 00 FF 25 4C 30 40 00    ...j.......%L0@.
0620   FF 25 54 30 40 00 00 00-00 00 00 00 00 00 00 00    .%T0@...........
```

4. 资源节

资源节一般名为.rsrc。这个节放有如图标、对话框等程序要用到的资源。

资源节是树形结构的，它有一个主目录，主目录下又有子目录，子目录下可以是子

目录或数据。根目录或子目录都是一个 IMAGE_RESOURCE_DIRECTORY 结构，如表 9-8 所示。

表 9-8 IMAGE_RESOURCE_DIRECTORY 结构

顺序	名字	大小（字节）	描述
1	Characteristics	4	通常为 0
2	TimeDateStamp	4	资源生成时间
3	MajorVersion	2	主版本号
4	MinorVersion	2	次版本号
5	NumberOfNamedEntries	2	以名字标识的资源数
6	NumberOfIdEntries	2	以 ID 标识的资源数

这个结构后面会跟着一个名为 DirectoryEntries，类型为 IMAGE_RESOURCE_DIRECTORY_ENTRY 的结构数组，数组的项数是表 9-8 中的 NumberOfNamedEntries 和 NumberOfIdEntries 的和。即是以名字和 ID 标识的资源的和。因为要表示用名字和 ID 标识的资源，因此 IMAGE_RESOURCE_DIRECTORY_ENTRY 结构有些复杂，它的 C 结构定义如下：

```
typedef struct _IMAGE_RESOURCE_DIRECTORY_ENTRY {
    union {
        struct {
            DWORD NameOffset:31;
            DWORD NameIsString:1;
        };
        DWORD    Name;
        WORD     Id;
    };
    union {
        DWORD    OffsetToData;
        struct {
            DWORD    OffsetToDirectory:31;
            DWORD    DataIsDirectory:1;
        };
    };
} IMAGE_RESOURCE_DIRECTORY_ENTRY;
```

下面是 Tdump.exe 对第三章中 Asm32.exe 的资源的分析：

```
Resources:
Type          Name                  Lang Id
-------------------------------------------
[0 named entries, 3 ID entries]
3   (next directory @00000028)
```

```
            [0 named entries, 1 ID entries]
            1   (next directory @00000070)
                                    [0 named entries, 1 ID entries]
                                    0   (data @000000B8)
                                        Offset:     00005108
                                        Size:       000002E8
                                        Code Page: 00000000
                                        Reserved:   00000000
5   (next directory @00000040)
            [1 named entries, 0 ID entries]
            MYDIALOG    (next directory @00000088)
                                    [0 named entries, 1 ID entries]
                                    0   (data @000000C8)
                                        Offset:     000053F0
                                        Size:       000000AE
                                        Code Page: 00000000
                                        Reserved:   00000000
14  (next directory @00000058)
            [1 named entries, 0 ID entries]
            MYICON  (next directory @000000A0)
                                    [0 named entries, 1 ID entries]
                                    0   (data @000000D8)
                                        Offset:     000054A0
                                        Size:       00000014
                                        Code Page: 00000000
                                        Reserved:   00000000
```

资源节更具体的结构相当复杂，这是只是一个初步的说明而已。

5. 引入函数节

这个节一般名为.idata，它包含有从其它（系统的或第三方写的）DLL 中引入的函数，例如 USER32.dll、GDI32.DLL 等。引入函数节也叫引入表，它的开始是一个 IMAGE_IMPORT_DESCRIPTOR 数组。这个数组的长度不定，但它的最后一项是全 0，可以依此判断数组的结束。IMAGE_IMPORT_DESCRIPTOR 的各项如表 9-9 所示。

表 9-9 IMAGE_IMPORT_DESCRIPTOR

顺序	名字	大小（字节）	描述
1	OriginalFirstThunk	4	IMAGE_THUNK_DATA 数组的指针
2	TimeDateStamp	4	文件建立时间
3	ForwarderChain	4	一般为 0
4	Name	4	DLL 名字的指针
5	FirstThunk	4	通常也是 IMAGE_THUNK_DATA 数组的指针

说明：

1）OriginalFirstThunk 是一个 IMAGE_THUNK_DATA 数组的 RVA，如下所示的 0A00H 处的 RVA 是 303CH，又由于本节的 RVA 是 3000H，因此在本文件中这个数组在 0A3CH 处。

2）TimeDateStamp，文件建立时间一般为 0。

3）ForwarderChain，这是当作一条链，当你的程序引用一个 DLL 中的 API，而这个 API 又引用别的 DLL 的 API 时用。但很少会有这样的例子。

4）Name 是 DLL 名字的指针。这是个 RVA，例如下面 0AC0H 处显示这个 RVA 是 305C，即文件的 0A5CH 中，是一个字符串"USER32.dll"。

5）FirstThunk 有多种意义，但通常也是一个 IMAGE_THUNK_DATA 数组的 RVA 地址。

IMAGE_IMPORT_DESCRIPTOR 数组如下：

```
0A00   3C 30 00 00 00 00 00 00-00 00 00 00 5C 30 00 00   <0..........\0..
                (1)         (2)         (3)         (4)
0A10   4C 30 00 00 44 30 00 00-00 00 00 00 00 00 00 00   L0..D0..........
                (5)         (1)         (2)         (3)
0A20   67 30 00 00 54 30 00 00-00 00 00 00 00 00 00 00   g0..T0..........
                (4)         (5)         (1)         (2)
0A30   00 00 00 00 00 00 00 00-00 00 00 00                ............
                (3)         (4)         (5)
```

IMAGE_THUNK_DATA 是一个有多种意义的类型，但是，它通常是指向一个名为 IMAGE_IMPORT_BY_NAME 的数组的 RVA 地址。

下面是 USER32.dll 和 KERL32.dll 所使用的 OriginalFirstThunk 成员和 FirstThunk 成员所指向的 IMAOE_THUNK_DATA 数组。USER32.dll 的 OriginalFirstThunk 和 FirstThunk 分别在 0A3CH 和 0A4CH，而 KERNEL32.dll 的则在 0A44H 和 0A54H。这两组 IMAGE_THUNK_DATA 中的第二组在装入内存时会改为实际 API 的地址。

```
0A30                                      74 30 00 00   t0..
0A40   00 00 00 00 82 30 00 00-00 00 00 00 74 30 00 00   .....0......t0..
0A50   00 00 00 00 82 30 00 00-00 00 00 00               .....0......
```

下面是 Name 指出的 DLL 的名字：

```
0A50                                      55 53 45 52   USER
0A60   33 32 2E 64 6C 6C 00 4B-45 52 4E 45 4C 33 32 2E   32.dll.KERNEL32.
0A70   64 6C 6C 00                                        dll.
```

IMAGE_IMPORT_BY_NAME 有两个成员，如表 9-10 所示。

表 9-10 IMAGE_IMPORT_BY_NAME

顺序	名字	大小（字节）	描述
1	Hint	2	索引
2	Name[]	不定	API 名字

说明：

1）Hint 估计是函数的引出号，在我们的例子中这个字段都是 0。

2）Name 是所引用的 API 的名字。

下面是 IMAGE_IMPORT_BY_NAME 数组：

```
0A70                00 00 4D 65-73 73 61 67 65 42 6F 78      ..MessageBox
0A80  41 00 00 00 45 78 69 74-50 72 6F 63 65 73 73 00      A...ExitProcess.
```

6. 引出函数节

这个节一般名为.edata，这是本文件向其它程序提供调用的函数列表。这个节一般用在 DLL 中，.EXE 文件也可以有这个节，但通常很少使用。

引出函数节的开头是一个 IMAGE_EXPORT_DIRECTORY 结构，如表 9-11 所示。

表 9-11 IMAGE_EXPORT_DIRECTORY 结构

顺序	名字	大小（字节）	描述
1	Characteristics	4	一般为 0
2	TimeDateStamp	4	文件生成时间
3	MajorVersion	2	主版本号
4	MinorVersion	2	次版本号
5	Name	4	指向 DLL 的名字
6	Base	4	开始的序列号
7	NumberOfFunctions	4	AddressOfFunctions 数组的项数
8	NumberOfNames	4	NumberOfNames 数组的项数
9	AddressOfFunctions	4	指向函数地址数组
10	AddressOfNames	4	函数名字的指针的地址
11	AddressOfNameOrdinals	4	指向输出序列号数组

说明：

1）第 7、8 项一般说来是相等的。

2）第 9 项指出了函数的 RAV 地址列表的 RVA 地址。

3）第 10 项放着一个地址，在这个地址中放有引出函数的名字的地址。

4）第 11 项是输出函数的序列号数组的指针，数组每一项长为 2 字节。

下面是本书前面的文件 DllDemo.dll 中的引出函数节。这个 DLL 只有一个引出函数 MsgBox。

```
0C00  00 00 00 00 00 00 00 00-00 00 00 00 32 40 00 00   ............2@..
          (1)         (2)       (3)  (4)     (5)
0C10  01 00 00 00 01 00 00 00-01 00 00 00 28 40 00 00   ............(@..
            (6)        (7)        (8)        (9)
0C20  2C 40 00 00 30 40 00 00-08 10 00 00 3E 40 00 00   ,@..0@......>@..
            (10)       (11)
0C30  00 00 44 6C 6C 44 65 6D-6F 2E 44 4C 4C 00 4D 73   ..DllDemo.DLL.Ms
```

0C40 67 42 6F 78 00 00 00 00-00 00 00 00 00 00 00 00 gBox...........

7. 重定位表节

存放有一个重定位表。若装载器不是把程序装到程序编译时默认的基地址时，就需要这个重定位表来作一些调整。

重定位节以 IMAGE_BASE_RELOCATION 结构开始，如表 9-12 所示。

表 9-12 IMAGE_BASE_RELOCATION 结构

顺序	名字	大小（字节）	描述
1	VirtualAddress	4	重定位数据开始 RVA 地址
2	SizeofBlock	4	这一块的大小
3	TypeOffset[]	不定	重定项位数组

VirtualAddress 是这一组重定位数据的开始 RVA 地址。下面指出的各重定位项的地址加上这个值才是该重定位项完整的 RVA 地址。这个地址是一个 4KB（1 页）的边界，PE 文件为每一个要重定位的页面生成一个 IMAGE_BASE_RELOCATION 结构块。

1）SizeofBlock 是这一结构块的大小。因为 1 项和 2 项大小都是固定的 4 字节，因此这个项减去 8 则是第 3 项的大小。

2）TypeOffset 是一个数组，数组中每一项大小是 2 字节。它要重定位的数据的位置。其低 12 位（12 位刚好表示 1 页）加上 VirtualAddress 即是要重定位的数据的 RVA 地址。

那么，如何决定重定位项要填入的数值呢？实际上，PE 文件生成时，已为文件的装入地址设置了一个默认值，要重定位的项中已有了为这个默认值设定的初始值。如果装入时按这个默认值作基地址装入，则不需再重定位，否则，把实际装入的地址与默认值的差加到要重定位的数据中去即可。

Hello.exe 的重定位表如下，可看到有 4 个重定位项。为什么是 4 个？我们看看它的反汇编结果就明白了：

```
001B:00401000   PUSH    00
001B:00401002   PUSH    00402000
001B:00401007   PUSH    0040200D
001B:0040100C   PUSH    00
001B:0040100E   CALL    USER32!MessageBoxA
001B:00401013   PUSH    00
001B:00401015   CALL    KERNEL32!ExitProcess
001B:0040101A   JMP     [USER32!MessageBoxA]
001B:00401020   JMP     [KERNEL32!ExitProcess]
```

重定位是在 push 两个字符串的针的地方和两个 jmp 指令的地方。

0C00 00 10 00 00 **14 00 00 00**-03 30 08 30 1C 30 22 30 ....0.0.0"0

(1) (2) (3)

8. 线程局部存储节

线程局部存储节名字一般叫.tls。若你用 C/C++编程，则在声明一个变量时，如果在

前面加上_declspec(thread)，如下所示：

_decspec(thread) int i;

在程序运行时每一个线程都会有这个变量的一个拷贝。Windows 有几个 API 可以代替这项工作：

TlsAlloc、TlsFree、TlsSetValue、TlsGetValue。

9.2 PE 文件查看器

下面介绍几个常用的 PE 文件查看器。

9.2.1 Tasm 的文件查看器

Tasm 提供一个 PE 文件查看器叫 Tdump.exe，它能分析出 PE 文件的特征。我们来看用 Tdump 对 Hello.exe 的查看结果：

```
Turbo Dump  Version 4.2.16.1 Copyright (c) 1988, 1996 Borland International
                    Display of File HELLO.EXE
Old Executable Header
DOS File Size                                   1000h  (  4096. )
Load Image Size                                 210h   (   528. )
Relocation Table entry count                    0000h  (     0. )
Relocation Table address                        0040h  (    64. )
Size of header record       (in paragraphs)     0004h  (     4. )
Minimum Memory Requirement (in paragraphs)      000Fh  (    15. )
Maximum Memory Requirement (in paragraphs)         FFFFh  ( 65535. )
File load checksum                                 0000h  (     0. )
Overlay Number                                     001Ah  (    26. )
Initial Stack Segment    (SS:SP)           0000:00B8
Program Entry Point      (CS:IP)           0000:0000
Portable Executable (PE) File
Header base: 00000100
CPU type                        80386
Flags                           818E [ executable 32bit ]
DLL flags                       0000 [ ]
Linker Version                  2.19
Time stamp                      B85C2775
O/S Version                     1.0
User Version                    0.0
Subsystem Version               3.10
Subsystem                       0002 [ Windows GUI ]
```

```
Object count                00000004
Symbols offset              00000000
Symbols count               00000000
Optional header size        00E0
Magic #                     10B
Code size                   00000200
Init Data size              00000600
Uninit Data size            00000000
Entry RVA                   00001000
Image base                  00400000
Code base                   00001000
Data base                   00002000
Object/File align           00001000/00000200
Reserved                    00000000
Image size                  00005000
Header size                 00000400
Checksum                    00000000
Stack reserve/commit        00100000/00002000
Heap reserve/commit         00100000/00001000
Number interesting RVAs  00000010
Name                        RVA          Size
------------------   ------------   ------------
Exports                  00000000   00000000
Imports                  00003000   00000090
Resources                00000000   00000000
Exceptions               00000000   00000000
Security                 00000000   00000000
Fixups                   00004000   00000014
Debug                    00000000   00000000
Description              00000000   00000000
TLS                      00000000   00000000
Callbacks                00000000   00000000
Reserved                 00000000   00000000
Reserved                 00000000   00000000
Reserved                 00000000   00000000
Reserved                 00000000   00000000
Reserved                 00000000   00000000
Object table:
```

```
#    Name        VirtSize    RVA         PhysSize    Phys off    Flags
---  --------    ----------- ----------- ----------- ----------- -----------
01   CODE    00001000    00001000    00000200    00000600    60000020
[CER]
02   DATA    00001000    00002000    00000200    00000800    C0000040
[IRW]
03   idata00001000    00003000    00000200    00000A00    C0000040 [IRW]
04   reloc00001000    00004000    00000200    00000C00    50000040 [ISR]
Key to section flags:
  C - contains code
  E - executable
  I - contains initialized data
  R - readable
  S - shareable
  W - writeable
Imports from USER32.dll
    MessageBoxA
Imports from KERNEL32.dll
    ExitProcess
```

这个程序详细分析了 PE 文件的内容，你可以在里面找到大多数所感兴趣的部分。

9.2.2 Windows "快速查看"

另外，有一个较容易找到的 PE 文件查看器是 Windows "快速查看"。这个程序在 Windows 95/98 中提供，但这是 Windows 安装的可选项，您的机器里可能没有这个程序。如果有安装的话，那么在右击应用程序的图标时，在弹出菜单中会有"快速查看"的选项，如图 9-5 所示。

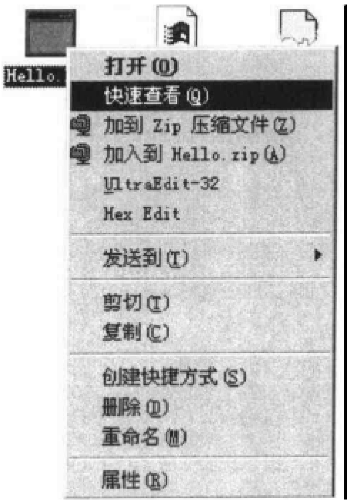


图 9-5

由于“快速查看”是 Microsoft 随 Windows 提供的程序，因此这个程序应该说更具权威性。“快速查看”对 Hello.exe 的分析结果如图 9-6 所示。

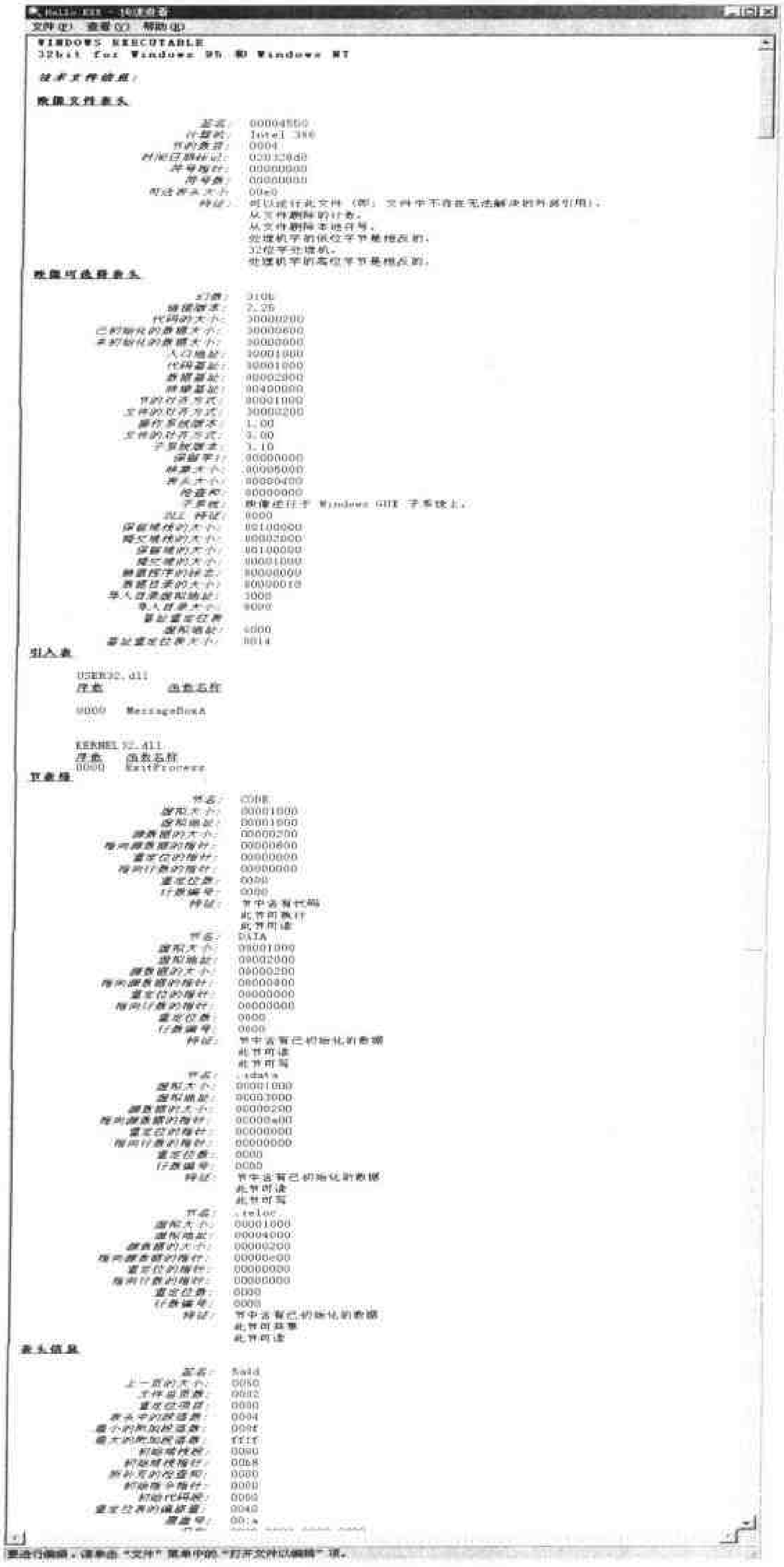


图 9-6

9.2.3 PEDump 文件查看器

Matt Pietrek 先生编写的 PE 文件查看器 PEDump.exe 也是一个很好的文件查看器。它能列出更详细的文件信息，并可以把对应的节按 16 进制打印出来，这是一个很好的分析工具。下面是 PEDump.exe 对 Asm32.exe 的分析：

```
F:\Bk\Chapter3\Asm32>pedump asm32.exe
Dump of file ASM32.EXE

File Header
  Machine:                      014C (i386)
  Number of Sections:           0005
  TimeDateStamp:                A8872902
  PointerToSymbolTable:         00000000
  NumberOfSymbols:              00000000
  SizeOfOptionalHeader:         00E0
  Characteristics:              818E
    EXECUTABLE_IMAGE
    LINE_NUMS_STRIPPED
    LOCAL_SYMS_STRIPPED
    BYTES_REVERSED_LO
    32BIT_MACHINE
    BYTES_REVERSED_HI

Optional Header
  Magic                         010B
  linker version                2.25
  size of code                  200
  size of initialized data      C00
  size of uninitialized data    0
  entrypoint RVA                1000
  base of code                  1000
  base of data                  2000
  image base                    400000
  section align                      1000
  file align                    200
  required OS version           1.00
  image version                 0.00
  subsystem version             3.10
```

```
    Reserved1                 0
    size of image             6000
    size of headers           400
    checksum                  0
    Subsystem                 0002 (Windows GUI)
    stack reserve size        100000
    stack commit size         2000
    heap reserve size         100000
    heap commit size          1000
    RVAs & sizes              10
  Data Directory
    EXPORT          rva: 00000000   size: 00000000
    IMPORT          rva: 00003000   size: 000000F4
    RESOURCE        rva: 00005000   size: 00000600
    EXCEPTION       rva: 00000000   size: 00000000
    SECURITY        rva: 00000000   size: 00000000
    BASERELOC       rva: 00004000   size: 00000020
    DEBUO           rva: 00000000   size: 00000000
    COPYRIGHT       rva: 00000000   size: 00000000
    GLOBALPTR       rva: 00000000   size: 00000000
    TLS             rva: 00000000   size: 00000000
    LOAD_CONFIG     rva: 00000000   size: 00000000
    BOUND_IMPORT    rva: 00000000   size: 00000000
    IAT             rva: 00000000   size: 00000000
    Unused                rva: 00000000   size: 00000000
    Unused                rva: 00000000   size: 00000000
    Unused                rva: 00000000   size: 00000000

  Section Table
    01 CODE       VirtSize: 00001000   VirtAddr:   00001000
      raw data offs:  00000600   raw data size:   00000200
      relocation offs:00000000   relocations:     00000000
      line # offs:    00000000   line #'s:        00000000
      characteristics:     60000020
        CODE   MEM_EXECUTE   MEM_READ

    02 DATA       VirtSize: 00001000   VirtAddr:     00002000
      raw data offs:       00000800   raw data size:   00000200
```

```
    relocation offs:   00000000  relocations:    00000000
    line # offs:       00000000  line #s:        00000000
    characteristics:   C0000040
      INITIALIZED_DATA  MEM_READ  MEM_WRITE

  03 .idata    VirtSize: 00001000  VirtAddr:     00003000
    raw data offs:     00000A00  raw data size:  00000200
    relocation offs:   00000000  relocations:    00000000
    line # offs:       00000000  line #s:        00000000
    characteristics: C0000040
      INITIALIZED_DATA  MEM_READ  MEM_WRITE

  04 .reloc    VirtSize:     00001000  VirtAddr:     00004000
    raw data offs:     00000C00  raw data size:      00000200
    relocation offs:   00000000  relocations:        00000000
    line # offs:       00000000  line #s:            00000000
    characteristics: 50000040
      INITIALIZED_DATA  MEM_SHARED  MEM_READ

  05 .rsrc     VirtSize:     00001000  VirtAddr:     00005000
    raw data offs:           00000E00  raw data size:  00000600
    relocation offs:         00000000  relocations:    00000000
    line # offs:             00000000  line #s:        00000000
    characteristics:         50000040
      INITIALIZED_DATA  MEM_SHARED  MEM_READ

Resources
ResDir (0) Named:00 ID:03 TimeDate:392CAB68 Vers:0.00 Char:0
    ResDir (ICON) Named:00 ID:01 TimeDate:392CAB68 Vers:0.00 Char:0
        ResDir (1) Named:00 ID:01 TimeDate:392CAB68 Vers:0.00 Char:0
            ID: 00000000  DataEntryOffs: 000000B8
            Offset: 05108  Size: 002E8  CodePage: 0
    ResDir (DIALOG) Named:01 ID:00 TimeDate:392CAB68 Vers:0.00 Char:0
        ResDir (MYDIALOG) Named:00 ID:01 TimeDate:392CAB68 Vers:0.00 Char:0
            ID: 00000000  DataEntryOffs: 000000C8
            Offset: 053F0  Size: 000AE  CodePage: 0
    ResDir (GROUP_ICON) Named:01 ID:00 TimeDate:392CAB68 Vers:0.00 Char:0
        ResDir (MYICON) Named:00 ID:01 TimeDate:392CAB68 Vers:0.00 Char:0
```

```
            ID: 00000000   DataEntryOffs: 000000D8
            Offset: 054A0   Size: 00014   CodePage: 0
Imports Table:
  USER32.dll
  Hint/Name Table: 0000303C
  TimeDateStamp:    00000000
  ForwarderChain:   00000000
  First thunk RVA: 0000305C
  Ordn   Name
     0   SetDlgItemTextA
     0   DialogBoxParamA
     0   EndDialog
     0   MessageBoxA

  KERNEL32.dll
  Hint/Name Table: 00003050
  TimeDateStamp:    00000000
  ForwarderChain:   00000000
  First thunk RVA: 00003070
  Ordn   Name
     0   GetModuleHandleA
     0   ExitProcess
```

附 录

A.1 系统环境设置

本书的 DOS 程序用 Tasm 5.0 编译，一般 Windows 程序用 Tasm 5.0 或 Visual C++ 6.0 编译，而 VxD 程序用 98DDK 和 Visual C++ 6.0 编译，因此，要编译本书的程序，必须先安装和设置好这 3 个编译环境。

这几个编译器的安装都比较直观，不需要太多介绍，只要依照安装程序的提示操作即可。这里只介绍设置的方法。

Tasm 基本上不需要怎么设置，只是如果用户想使用方便，应该将 Tasm 的 bin 目录放在搜索路径中。例如，如果用户 Tasm 安装在路径 c:\tasm，则在 autoexec.bat 文件的后面加上如下一句：

SET path=%PATH%;c:\tasm\bin

Visual C++在安装时会生成一个环境设置文件 Vcvars.bat，并放在 Visual C++的 bin 目录下，在本书作者的机器上，它的全路径是：C:\Program Files\Microsoft Visual Studio\VC98\Bin\vcvars.bat。如果是在 Windows NT 上安装 Visual C++，则安装程序会为用户设置好环境变量，但在 Windows 98 中安装时，却要根据这个 bat 文件来手工设置环境变量。98DDK 与此相似，在它的 bin 目录下有一个环境设置的批处理文件 Setenv.bat 文件，在本书作者的机器上，它的全路径是 C:\98DDK\bin\setenv.bat，读者也是要手工设置它。

这两个 bat 文件主要设置了几个环境变量：PATH、LIB 和 INCLUDE 等。用户可以在 C 盘根目录的 autoexec.bat 文件中设置这些环境变量。下面本书作者给出其机器上的 autoexec.bat 中这几个相关变量的设置为例子，读者可以根据它和自己机器的实际情况来修改：

```
SET path=C:\PWIN98;C:\PWIN98\COMMAND;c:\98ddk\bin;c:\98ddk\bin\win98;c:\tools;e:\language\tasm\bin;c:\masm32\bin;C:\Progra~1\Micros~1\VC98\Bin;C:\Progra~1\Micros~1\Common\MSDev98\Bin
set VSCommonDir=D:\PROGRA~1\MICROS~1\Common
set MSDevDir=D:\PROGRA~1\MICROS~1\Common\msdev98
set MSVCDir=c:\PROGRA~1\MICROS~1\VC98
set INCLUDE=%MSVCDir%\ATL\INCLUDE;%MSVCDir%\INCLUDE;%MSVCDir%\MFC\INCLUDE;%INCLUDE%
set LIB=%MSVCDir%\LIB;%MSVCDir%\MFC\LIB;%LIB%
set LIB=%LIB%;C:\98DDK\lib\i386\free;C:\98DDK\lib\i386\checked
set INCLUDE=%INCLUDE%;C:\98DDK\inc\win98
```

A.2 用 Masm 编写 Windows 程序

98DDK 中提供的汇编程序编译器也可以用于编写一般的 Windows 程序，这是一个不错的汇编程序编译器，利用它，可以写出更易懂、更结构化的汇编程序。

但是，使用这个编译器编写汇编程序的一个麻烦是没有所需的库文件和头文件，必须要自己来生成。不过不用担心，现在用户不必这样做了，因为已有人做了这个工作。互联网上有一个称为 Masm32 的软件包，它将使用 Masm 编程所需的各种工具、库文件、头文件、甚至编辑器都集成一起了，用它来写汇编程序非常方便。

下面是一个用 Masm32 写的简单 Windows 程序，但它已包括了一个通用的 Windows 程序的框架：注册窗口类、建立窗口、建立消息循环，程序有对菜单消息的响应，对按钮控制的消息响应，有自建的窗口和用对话框模板建立的对话框。程序运行结果如图 A-1 所示。

图 A-1

下面是源文件 Sample.asm。文件用关键字 include 和 includelib 指示编译器包含头文件和连接时要用的库文件，这样在连接时就不需要再指定系统 DLL 对应的库文件。

程序使用了一个技巧，它定义了一个宏 szText，这个宏的作用是让用户的程序可以在代码段的任意位置定义一个字符串。这个宏很有用，它使得用户可以在一个子程序的内部而不是全局的数据段中定义一个字符串，这有利于程序的结构化。但要注意的是这个字符串是放在代码段中的，因此，用户不要尝试去写它。

在两个消息处理函数中，使用了.if、.elseif 和.endif 指令，避免了大量的 cmp、jz 指令对，大大地加强了程序的可读性（读者也可以在 Tasm 中使用这个技巧）。

```
;***********************************
;文件：Sample.asm                  *
;功能：用 Masm 写 Win32 程序的     *
;        简单例子                  *
;***********************************
        .386
        .model flat, stdcall
```

```
        option casemap :none

        include c:\masm32\include\windows.inc          ;头文件
        include c:\masm32\include\gdi32.inc
        include c:\masm32\include\user32.inc
        include c:\masm32\include\kernel32.inc
        includelib c:\masm32\lib\gdi32.lib             ;库文件
        includelib c:\masm32\lib\user32.lib
        includelib c:\masm32\lib\kernel32.lib

        szText MACRO Name, Text:VARARG                 ;宏定义
            LOCAL lbl
            jmp lbl
            Name db Text,0
            lbl:
        ENDM

        IDM_MAINMENU= 600                              ;常量定义
        IDM_EXIT    = 601
        IDM_ABOUT = 602
        IDI_MYICGN= 700
        IDD_ABOUT = 1000

        WinMain PROTO :DWORD,:DWORD,:DWORD,:DWORD      ;函数原型
        WndProc PROTO :DWORD,:DWORD,:DWORD,:DWORD
        AboutDlgProc PROTO :DWGRD,:DWORD,:DWORD,:DWORD

    .DATA                                              ;数据
        hInstance dd 0
        CommandLine dd 0

    .CODE                                              ;代码
    start:
        invoke  GetModuleHandle, NULL                  ;取模块句柄
        mov     hInstance, eax

        invoke  GetCommandLine                         ;取命令行
        mov     CommandLine, eax
```

```
    invoke    WinMain,hInstance,NULL,CommandLine,SW_SHOWDEFAULT

    invoke    ExitProcess,eax                    ;退出进程
;*****************************************************************
;WinMain 函数
WinMain    PROC    hInst:DWORD,    hPreInst:DWORD,    szCmdLine:DWORD,
nCmdShow:DWORD
        LOCAL  wc  :WNDCLASSEX
        LOCAL  msg :MSG
        LOCAL  hMainWin :DWORD

        szText szClassName,"my_win_class"          ;窗口类名
        szText szTitle, "Masm Sample"              ;窗口标题

        mov     wc.cbSize,     SIZEOF WNDCLASSEX
        mov     wc.style,  CS_HREDRAW or CS_VREDRAW
        mov     wc.lpfnWndProc,       OFFSET WndProc
        mov     wc.cbClsExtra,        NULL
        mov     wc.cbWndExtra,        NULL
        mov     eax,            hInst
        mov     wc.hInstance,        eax
        invoke  GetStockObject,   WHITE_BRUSH
        mov     wc.hbrBackground, eax
        mov     wc.lpszMenuName, NULL
        mov     wc.lpszClassName, OFFSET szClassName
        invoke  LoadIcon,hInst,   IDI_MYICON
        mov     wc.hIcon, eax
        invoke  LoadCursor,NULL,IDC_ARROW
        mov     wc.hCursor,   eax
        mov     wc.hIconSm,  0

        invoke  RegisterClassEx,ADDR wc            ;注册窗口类

        invoke  GetSystemMetrics,SM_CXSCREEN   ;取屏幕宽
        push    eax
        invoke  GetSystemMetrics,SM_CYSCREEN   ;取屏幕高
        pop     ebx
        shr     ebx,1                          ;宽除以 2
```

```
        shr     eax,1                           ;高除以 2

        invoke  CreateWindowEx,0,               ;建立主窗口
                        ADDR szClassName,
                        ADDR szTitle,
                        WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        ebx,eax,
                        NULL,NULL,
                        hInst,NULL
        mov hMainWin,eax                        ;窗口句柄

        invoke  ShowWindow,hMainWin,SW_SHOWNORMAL
        invoke  UpdateWindow,hMainWin

    MsgLoop:                                    ;消息循环
        invoke  GetMessage,ADDR msg,NULL,0,0
        or      eax,eax
        jz      Exit
        invoke  TranslateMessage, ADDR msg
        invoke  DispatchMessage,  ADDR msg
        jmp     MsgLoop

    Exit:
        mov     eax,msg.wParam
        ret
    WinMain ENDP
    ;*************************************************************
    ;主窗口过程
    WndProc PROC hWnd:DWORD, wMsg:DWORD, wParam:DWORD, lParam:DWORD

        .if wMsg == WM_COMMAND
            .if wParam == IDM_EXIT                  ;用户选择菜单项“退出”
            invoke  SendMessage,hWnd,WM_SYSCOMMAND,SC_CLOSE,NULL
            .elseif wParam == IDM_ABOUT             ;用户选择菜单项“关于”
                invoke  DialogBoxParamA,hInstance,IDD_ABOUT,hWnd,OFFSET
AboutDlgProc,0
            .endif
```

```
        .elseif wMsg == WM_CREATE
            invoke    LoadMenu,hInstance,IDM_MAINMENU
            invoke    SetMenu,hWnd,eax
        .elseif wMsg == WM_DESTROY
            invoke    PostQuitMessage,NULL
            xor  eax,eax
            ret
        .endif

        invoke DefWindowProc,hWnd,wMsg,wParam,lParam
        ret

    WndProc ENDP
    ;*************************************************************
    ; "关于" 对话框处理过程
    AboutDlgProc    PROC        hWnd:DWORD,    wMsg:DWORD,    wParam:DWORD,
lParam:DWORD

        .if wMsg == WM_COMMAND
            .if wParam == IDOK
                invoke    EndDialog,hWnd,1
            .endif

        .elseif wMsg == WM_CLOSE
            invoke    EndDialog,hWnd,1
        .endif

        xor  eax,eax
        ret

    AboutDlgProc ENDP
    ;*************************************************************
        end start
```

下面是资源文件 Rc.rc。这个文件定义了主程序的菜单和"关于"对话框。

```
#include "c:\masm32\include\resource.h"
#define IDM_MAINMENU   600
#define IDM_EXIT 601
#define IDM_ABOUT   602
```

```
#define IDI_MYICON   700
#define IDD_ABOUT    1000

IDI_MYICON ICON DISCARDABLE "MyIcon.ICO"
IDM_MAINMENU MENUEX DISCARDABLE
{
    POPUP "文件(&F)", , , 0
    {
        MENUITEM "退出(&E)", IDM_EXIT
    }
    POPUP "帮助(&H)", , , 0
    {
        MENUITEM "关于(&A)...", IDM_ABOUT
    }
}

IDD_ABOUT DIALOGEX 0, 0, 115,67
STYLE WS_DLGFRAME|WS_CAPTION|WS_SYSMENU|DS_CENTER
CAPTION "关于"
FONT 10,"隶书"

BEGIN
   CTEXT "用 Masm 写 Windows 程序",-1,3,18,111,8
   PUSHBUTTON "确定",IDOK,40,47,40,14
END
```

可以写一个 Makefile 文件来帮助编译这个程序，但本例要编译的程序很简单，这里只写了一个批处理文件 make.bat：

```
c:\masm32\bin\ml -c -coff   sample.asm
c:\masm32\bin\rc   rc.rc
c:\masm32\bin\link   sample.obj   rc.res
```

A.3 Pentium 汇编指令

1. AAA 加法的 ASCII 码调整

示例：AAA

2. AAD 除法的 ASCII 码调整

示例：AAD

3. AAM 乘法的 ASCII 码调整
示例：AAM

4. AAS 减法的 ASCII 码调整
示例：AAS

5. ADC 带进位加
示例：ADC EAX,EBX

6. ADD 加
示例：ADD EAX,EBX

7. AND 逻辑与
示例：AND EAX,EBX

8. ARPL 调整请求特权级
示例：ARPL AX,BX ;如 AX 的 RPL 小于 BX 的 RPL，则改为 BX 的 RPL，且 ZF 置 1，否则 ZF 清 0

9. BOUND 越界检查
示例：BOUND AX,MEM ;如 AX 不在 MEM 及 MEM+2 中的值之间，则产生异常 5

10. BSF 从低向高扫描目标，查找对应为 1 的位
示例：BSF EAX,EBX ;把 EBX 由低到高第一个 1 的位置送 EAX，如全 0 则 ZF 置 1，否则 ZF 置 0

11. BSR 从高向低扫描目标，查找对应为 1 的位
示例：BSR EAX,EBX ;与 BSF 类似

12. BSWP 32 位寄存器高低字节交换
示例：BSWP EAX

13. BT 测试目标的指定位
示例：BT EAX,BL ;如 EAX 中 BL 指定的位为 1 则 CF=1，否则 CF=0

14. BTC 将目标指定位取反
示例：BTC EAX，3

15. BTR 将目标指定位清 0
示例：BTR EAX，3

16. BTS 将目标指定位置 1
示例：BTS EAX,3

17. CALL 调用
示例：CALL MyFunc

18. CBW AL 扩展到 AX
示例：CBW

19. CDQ EAX 扩展到 EDX-EAX
示例：CDQ

20. CLC 清除进位（CF）标志
示例：CLC

21. CLD 清除方向位（DF）标志
示例：CLD

22. CLI 清除中断允许位（IF）
示例：CLI

23. CLTS 清除任务开关标志（TSF）
示例：CLTS

24. CMC 进位标志取反
示例：CMC

25. CMP 比较大小
示例：CMP EAX,EBX

26. CMPSB 比较字符串，每次比较 1 个字节
示例：CMPSB

27. CMPSD 比较字符串，每次比较 4 个字节
示例：CMPSD

28. CMPSW 比较字符串，每次比较 2 个字节
示例：CMPSW

29. CMPXCHG 比较交换
示例：CMPXCHG BX,CX ;如果 AX 与 BX 相等，则 CX 送 BX 且 ZF 置 1，否则 BX 送 AX，且 ZF 清 0

30. CMPXCH8B 比较 EDX-EAX 与 64 位的目标，或若相等则 ECX-EBX 送目标且 ZF=1，否则目标送 EDX-EAX，ZF=0
示例：CMPXCH8B [ESI]

31. CPUID CPU 标识送 EAX、EBX、ECX、EDX
示例：CPUID

32. CWD 将 AX 扩展到 DX-AX
示例：CWD

33. CWDE 将 AX 扩展到 EAX
示例：CWDE

34. DAA 加法的 10 进制调整
示例：DAA

35. DAS 减法的 10 进制调整
示例：DAS

36. DEC 目标减 1
示例：DEC EAX

37. DIV 无符号减法
示例：DIV BL ;AX/BL，商在 AL 中，余数在 AH 中

38. ENTER 为子过程建立堆栈框架
示例：ENTER 12，1

39. ESC 处理器放弃总线
示例：ESC

40. HLT 处理器停机
示例：HLT

41. IDIV 带符号除
示例：IDIV BL

42. IMUL 带符号乘法
示例： IMUL BX
IMUL EAX,EBX,1000

43. IN 从端口输入
示例：IN AL,10H

44. INC 目标加 1
示例：INC EAX

45. INSB 从端口输入字符串
示例：INSB

46. INSD 从端口输入双字串
示例：INSD

47. INSW 从端口输入字串
示例：INSW

48. INT 中断
示例：INT 21H
INT 3

49. INTO 溢出则中断
示例：INTO

50. INVD 使处理器内部 CACHE 无效
示例：INVD

51. INVLPG 使 TLB 项无效
示例：INVLPG [EBX]

52. IRET 中断返回
示例：IRET

53. IRETD 中断返回
示例：IRETD

54. JA/JNBE 如 CF=0 且 ZF=0 则跳转
示例：JA LABEL1

55. JAE/JNB/JNC 如 CF=0 则跳转
示例：JAE LABEL1

56. JCXZ 如 CX 为 0 则跳转
示例：JCXZ LABEL1

57. JE/JZ 如 ZF=0 则跳转
示例：JE LABEL1

58. JECXZ 如 ECX 为 0 则跳转
示例：JECXZ LABEL1

59. JG/JNLE 如 ZF=0 且 SF=OF 则跳转
示例：JG LABEL1

60. JGE/JNL 如 SF=OF 则跳转
示例：JGE LABEL1

61. JL/JNGE 如 SF<>OF 则跳转
示例：JL LABEL1

62. JLE/JNG 如 ZF=1 或 SF<>OF 则跳转
示例：JLE LABEL1

63. JMP 无条件跳转
示例：JMP LABEL1

64. JNA/JBE 如 CF=1 或 ZF=1 则跳转
示例：JNA LABEL1

65. JNE/JNZ 如 ZF=0 则跳转
示例：JNE LABEL1

66. JNO 如 OF=0 则跳转
示例：JNO LABEL1

67. JNP/JPO 如 PF=0 则跳转
示例：JNP LABEL1

68. JNS 如 SF=0 则跳转
示例：JNS LABEL1

69. JO 如 OF=1 则跳转
示例：JO LABEL1

70. JP/JPE 如 PF=1 则跳转
示例：JP LABEL1

71. JS 如 SF=1 则跳转
示例：JS LABEL1

72. LAHF 标志寄存器低 8 位送 AH
示例：LAHF

73. LAR 将选择子的访问权送目的寄存器
示例：LAR AX,BX

74. LDS 地址送寄存器及 DS
示例：LDS EBX,MEM

75. LEA 取地址
示例：LEA EBX, LABEL1
LEA EBX,[ESI*4+EDI+4]

76. LEAVE 子过程返回释放堆栈框架
示例：LEAVE

77. LES 地址送寄存器及 ES

示例：LES EBX,MEM

78. LFS 地址送寄存器及 FS

示例：LFS EBX, LABEL1

79. LGDT 内存内容装入全局描述符表寄存器

示例：LGDT [EBP]

80. LGS 地址送寄存器及 GS

示例：LGS EAX,MEM

81. LIDT 内存内容装入中断描述符表寄存器

示例：LIDT [SI]

82. LLDT 装入局部描述符表寄存器

示例：LLDT BX

83. LMSW 装入机器状态字

示例：LMSW[BP]

84. LOCK 总线锁定

示例：LOCK ADD EAX,1

85. LODSB 把 DS：ESI 或 DS：SI 指向的内容送 AL

示例：LODSB

86. LODSD 把 DS：ESI 或 DS：SI 指向的内容送 AX

示例：LODSD

87. LODSW 把 DS：ESI 或 DS：SI 指向的内容送 EAX

示例：LODSW

88. LOOP 如 ECX-1＜＞0 则循环

示例：LOOP LABEL1

89. LOOPE/LOOPZ 如 ECX-1＜＞0 且 ZF=1 则循环

示例：LOOPE LABEL1

90. LOOPNE/LOOPNZ 如 ECX-1<>0 且 ZF=0 则循环
示例：LOOPNZ　LABEL1

91. LSL 装入选择子对应的描述符的段界限
示例：LSL　AX,BX　　;BX 对应的段的段界限送 AX

92. LSS 地址送寄存器及 FS
示例：LSS　ESP,MEM

93. LTR 装入任务寄存器
示例：LTR　AX

94. MOV 传送
示例：MOV EAX,12345678H

95. MOV 控制寄存器传送
示例：MOV CR0,EAX

96. MOV 调试寄存器传送
示例：MOV DR0,EAX
注：Pentium CPU 不支持测试寄存器（TRn）传送指令

97. MOVSB 串传送，字节为单位
示例：REP　MOVSB

98. MOVSD 串传送，双字为单位
示例：REP　MOVSD

99. MOVSW 串传送，字为单位
示例：REP　MOVSW

100. MOVSX 按符号扩展传送
示例：MOVSX　EAX,BX

101. MOVZX 按零扩展传送
示例：MOVZX　EAX,CL

102. MUL 无符号乘法

示例：MUL BL

103. NEG 对目标求补
示例：NEG AX

104. NOP 空操作
示例：NOP

105. NOT 对目标求反
示例：NOT EAX

106. OR 逻辑或
示例：OR AX,BX

107. OUT 输出到端口
示例：OUT 43H,AL

108. OUTSB 输出串到端口，字节为单位
示例：REPZ OUTSB

109. OUTSD 输出串到端口，双字为单位
示例：REPZ OUTSD

110. OUTSW 输出串到端口，字为单位
示例：REPZ OUTSW

111. POP 从堆栈弹出
示例：POP EAX

112. POPA 从堆栈弹出全部 16 位通用寄存器
示例：POPA

113. POPAD 从堆栈弹出全部 32 位通用寄存器
示例：POPAD

114. POPF 从堆栈弹出 16 位标志寄存器
示例：POPF

115. POPFD 从堆栈弹出 32 位标志寄存器
示例：POPFD

116. PUSH 数据进栈
示例：PUSH EAX

117. PUSHA 全部 16 位通用寄存器入栈
示例：PUSHA

118. PUSHAD 全部 32 位通用寄存器入栈
示例：PUSHAD

119. PUSHF 16 位标志寄存器进栈
示例：PUSHF

120. PUSHFD 32 位标志寄存器进栈
示例：PUSHFD

121. RCL 带进位循环左移
示例：RCL EAX,3

122. RCR 带进位循环右移
示例：RCR EAX,CL

123. RDMSR 把 ECX 指定的模型专用寄存器内容送 EDX-EAX
示例：RDMSR

124. RDTSC 读时间标记计数器到 EDX-EAX
示例：RDSC

125. REP/REPE/REPZ/REPNE/REPNZ 重复前缀
示例：REP MOVSB

126. RET 子过程返回
示例：RET

127. RET N 子过程返回并使 ESP 加 N
示例：RET 4

128. ROL 循环左移
示例：ROL EAX,4

129. ROR 循环右移
示例：ROR EAX,CL

130. RSM 从系统管理模式返回
示例：RSM

131. SAHF 送 AH 到标志寄存器
示例：SAHF

132. SAL/SHL 算术/逻辑左移
示例：SAL EBX,9

133. SAR 算术右移
示例：SAR ESI,CL

134. SBB 借位减法
示例：SBB EBX,EAX

135. SCASB 串扫描，以字节为单位
示例：REP SCASB

136. SCASD 串扫描，以双字为单位
示例：REP SCASD

137. SCASW 串扫描，以字为单位
示例：REP SCASW

138. SETA/SETNBE 如 CF=0 且 ZF=0 则置 1
示例：CMP BX,5
SETA AL

139. SETAE/SETNB/SETNC 如 CF=0 则置 1
示例：SETAE AL

140. SETB/SETC/SETNAE 如 CF=1 则置 1

示例：SETB EAX

141. SETBE/SETNA 如 CF=1 或 ZF=1 则置 1
示例：SETBE EAX

142. SETE/SETZ 如 ZF=1 则置 1
示例：SETE WORD PTR[ESI]

143. SETG/SETNLE 如 ZF=0 且 SF=OF 则置 1
示例：SETG WORD PTR[EBP]

144. SETGE/SETNL 如 SF=OF 则置 1
示例：SETGE EAX

145. SETL/SETNGE 如 SF<>OF 则置 1
示例：SETL EAX

146. SETLE/SETNG 如 SF<>OF 或 ZF=1 则置 1
示例：SETLEMEM

147. SETNE/SETNZ ZF=0 则置 1
示例：SETNE MEM

148. SETNO 如 OF=0 则置 1
示例：SETNO ESI

149. SETNP/SETPO 如 PF=0 则置 1
示例：SETNPCL

150. SETNS 如 SP=0 则置 1
示例：SETNSAL

151. SETO 如 OF=1 则置 1
示例：SETO EAX

152. SETP/SETPE 如 PF=1 则置 1
示例：SETP ESI

153. SETS 如 SF=1 则置 1
示例：SETS EAX

154. SGDT 保存全局描述符寄存器
示例：SGDT [EBP]

155. SHLD 双精度左移
示例：SHLD EAX,EBX,3
SHLD ESI,EDI,CL

156. SHR 逻辑右移
示例：SHR EAX,5

157. SHRD 双精度右移
示例：SHRD BX,AX,CL

158. SIDT 保存 IDT 寄存器
示例：SIDT [EBP]

159. SLDT 保存 LDT 选择子
示例：SLDT [EBX]

160. SMSW 保存机器状态字 MSW
示例：SMSW[EBP]

161. STC 进位标志置位
示例：STC

162. STD 方向标志置位
示例：STD

163. STI 中断标志置位
示例：STI

164. STOSB 串保存，以字节为单位
示例：STOSB ;AL 送 ES：EDI 指向的内存

165. STOSD 串保存，以双字为单位
示例：STOSD ;EAX 送 ES：EDI 指向的内存

166. STOSW 串保存，以字为单位

示例：STOSW ;AX 送 ES：EDI 指向的内存

167. STR 保存任务寄存器

示例：STR [EBP]

168. SUB 减法

示例：SUB EAX，ESI

169. TEST 测试，作“AND”运算，但只影响标志寄存器

示例：TEST EAX，EBX

170. VERR 如目标选择子对应段可读，则 ZF 置 1

示例：VERR BX ;如 BX 对应的段可读，则 ZF=1

171. VERW 如目标选择子对应段可写，则 ZF 置 1

示例：VERW BX ;如 BX 对应的段可写，则 ZF=1

172. WAIT 等待

示例：WAIT

173. WBINVD 写回内部 CACHE 并使之失效

示例：WBINVD

174. WRMSR 把 EDX-EAX 的内容写入 ECX 指定的模型专用寄存器

示例：WRMSR

175. XADD 目标和源相加，并使源保存目标的原值

示例：XADD EBX,EAX ;EAX 保存 EBX 的值，EBX 为原 EAX 与 EBX 的和

176. XCHG 交换数据

示例：XCHG EAX,EBX ;EAX 与 EBX 的值交换

177. XLAT/XLATB 以 BX 或 EBX 为基址，AL 为偏移地址换码

示例：XLAT

178. XOR 异或

示例：XOR EBX,EAX

A.4 MMX 指令和寄存器

支持 MMX 指令的计算机有 8 个 64 位的 MMX 寄存器，分别称作 XMM0~XMM7，但有些 MMX 指令也可以使用 32 的通用寄存器。XMM0~XMM7 实际上是 CPU 内置浮点单元 80X87 的寄存器 R0~R7 的别名，它的状态也使用 FPU 部件的状态，因此，如果要同时使用 MMX 指令和 80X87 指令，则要小心它们冲突。

MMX 技术提供单指令流多数据流（SIMD）的支持，即是说，可以一个指令同时操作多个数据。例如，指令 PADDSW 提供同时对 4 个字（16×4=64BIT）的带符号加法，一个指令即执行了用一般指令要用 4 个指令完成的操作。

SIMD 技术大大提高了多媒体的支持。如图像、声音等的解码，需要对大量的小数据进行相似的操作，而 SIMD 则使程序能对这些操作并行化。一个例子很多的声音数据是 16 位（1 个字）的音质，则可以通过对大量的 16 位数据进行并行操作而加快运算速度。

下面给出 MMX 指令及其使用示例。在 PII、PIII CPU 中，MMX 指令又有扩展，这里并没有全部给出。

1. EMMS　清空 MMX 状态

示例：EMMS

2. MOVD　双字传送

说明：实现通用寄存器与 MMX 寄存器之间、内存与 MMX 寄存器之间的双字传送

示例：
```
MOVD    XMM0 ,MEM_DWORD
MOVD    MEM_DWORD,XMM1
MOVD    XMM2,EAX
MOVD    EBX,XMM3
```

3. MOVQ　4 字传送

说明：实现 MMX 寄存器之间、内存与 MMX 寄存器之间的 4 字传送

示例：
```
MOVQ    XMM4,XMM5
MOVQ    XMM5,MEM_QUAD
```

4. PACKSSDW　紧缩带符号双字到字，溢出取饱和值

说明：把源和目的操作数的共 4 个带符号双字紧缩到目的操作数的 4 个带符号字，如果结果溢出，则取饱和值，如图 A-2 所示。

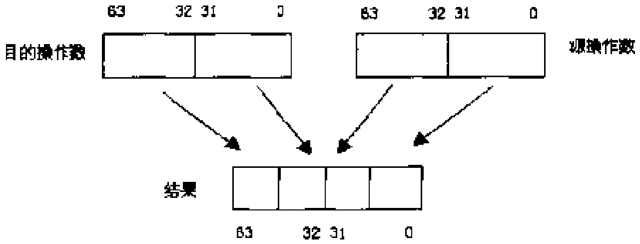


图 A-2

取饱和值——即取可以取的最大值或最小值。如本指令结果是带符号字，16 位，结果正数超过 7FFFH 或负数超过 8000H 时，则取 7FFFH 或 8000H。如果是无符号字，则最大、最小值是 FFFFH 和 0。

示例：PACKSSDW XMM0,XMM1
　　　PACKSSDW XMM0,MEM_QUAD

5. PACKSSWB　紧缩带符号字到字节，溢出取饱和值

说明：把源和目的操作数的共 8 个带符号字紧缩到目的操作数的 8 个带符号字节。如果结果溢出，则取饱和值。

请参考上一条指令。

示例：PACKSSWB XMM0,XMM1
　　　PACKSSWB XMM0,MEM_QUAD

6. PACKUSWB　紧缩无符号字到字节，溢出取饱和值

说明：把源和目的操作数的共 8 个带符号双字紧缩到目的操作数的 8 个无符号字节。如果结果溢出，则取饱和值，即大于 FFH 或小于 0，则结果取 FFH 或 0。

请参考上一条指令。

示例：PACKUSWB XMM0,XMM1
　　　PACKUSWB XMM0,MEM_QUAD

7. PADDB　紧缩字节相加，溢出则回绕

说明：源和目的操作数的 8 个字节分别相加，如结果溢出，则回绕。

回绕——即只取其能装入的部分。例如结果为 1C8H，因为字节只有 8 位，所以结果是 C8H。

示例：PADDB　XMM6,XMM7
　　　PADDB　XMM0,MEM_QUAD

8. PADDD　紧缩双字相加，溢出则回绕

说明：源和目的操作数的 2 个双字分别相加，如结果溢出，则回绕。

关于回绕，请参考第 7 条指令——PADDB。

示例：PADDD　XMM6,XMM7

9. PADDW　紧缩字节相加，溢出则回绕

说明：源和目的操作数的 4 个字节分别相加，如结果溢出，则回绕。

关于回绕，请参考第 7 条指令——PADDB

示例：PADDW　XMM6,XMM7

10. PADDSB　带符号紧缩字节相加，溢出取饱和值

说明：源和目的操作数的 8 个带符号字节分别相加，如结果溢出，则取饱和值。
关于饱和值，请参考第 4 条指令——PACKSSDW。
示例：PADDSB XMM1,XMM2
PADDSB XMM0,MEM_QUAD

11. PADDSW 带符号紧缩字相加，溢出取饱和值
说明：源和目的操作数的 4 个带符号字分别相加，如结果溢出，则取饱和值。
关于饱和值，请参考第 4 条指令——PACKSSDW。
示例：PADDSW XMM1,XMM2

12. PADDUSB 无符号紧缩字节相加，溢出取饱和值
说明：源和目的操作数的 8 个无符号字节分别相加，如结果溢出，则取饱和值。
关于饱和值，请参考第 4 条指令——PACKSSDW。
示例：PADDUSB XMM0,XMM1
PADDUSB XMM0,MEM_QUAD

13. PADDUSW 无符号紧缩字相加，溢出取饱和值
说明：源和目的操作数的 4 个无符号字分别相加，如结果溢出，则取饱和值。
关于饱和值，请参考第 4 条指令——PACKSSDW。
示例：PADDUSW XMM0,XMM1

14. PAND 逻辑与操作
说明：实现 MMX 寄存器之间、内存到 MMX 寄存器的逻辑与操作。
示例：PAND XMM4,XMM1
PAND XMM4,MEM_QUAD

15. PANDN 求反后与操作
说明：将源操作数求反后和目标操作数作与操作。允许 MMX 寄存器之间、内存到 MMX 寄存器的运算。
示例：PANDN XMM4,XMM1
PANDN XMM4,MEM_QUAD

16. PCMPEQB 紧缩字节相等比较
说明：实现 MMX 寄存器之间、内存到 MMX 寄存器之间的相等比较。
如果对应的字节相等，则将目的操作数对应的字节全部位设成 1，否则全部设成 0。
示例：PCMPEQB XMM0,XMM1
PCMPEQB XMM0,MEM_QUAD

17. PCMPEQD 紧缩双字相等比较

说明：参考第 16 条指令——PCMPEQB。

示例：PCMPEQD XMM0,XMM1

18. PCMPEQW 紧缩字相等比较

说明：参考第 16 条指令——PCMPEQB。

示例：PCMPEQW XMM0,XMM1

19. PCMPGTB 带符号紧缩字节大于比较

说明：实现 MMX 寄存器之间、内存到 MMX 寄存器之间的大于比较。

如果对应的字节目标操作数大于源操作数，则将目的操作数对应的字节全部位设成 1，否则全部设成 0。

示例：PCMPGTB XMM3,XMM1

PCMPGTB XMM2,MEM_QUAD

20. PCMPGTD 带符号紧缩双字大于比较

说明：参考第 19 条指令——PCMPGTB。

示例：PCMPGTD XMM3,XMM1

21. PCMPGTW 带符号紧缩字大于比较

说明：参考第 19 条指令——PCMPGTB。

示例：PCMPGTW XMM3,XMM1

22. PMADDWD 乘和加操作

说明：如图 A-3 所示，把源和目的操作数中的 2 个双字中的 2 个带符号字分别相乘后相加。实现 MMX 寄存器之间、内存到 MMX 寄存器之间的操作。

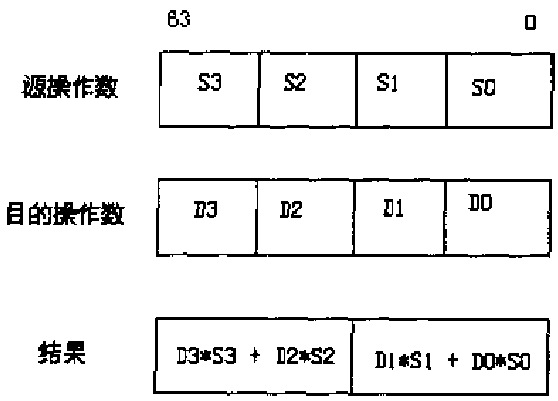


图 A-3

示例：PMADDWD XMM5,XMM2

PMADDWD XMM6,MEM_QUAD

23. PMULHUW 紧缩无符号字相乘，存高位

说明：源和目的操作数的 4 个无符号字分别相乘，并把结果的高 16 位存入目的寄存器相应字中。实现 MMX 寄存器之间、内存到 MMX 寄存器之间的操作。

示例：PMULHUW XMM3,XMM1

PMULHUW XMM3,MEM_QUAD

24. PMULHW 紧缩带符号字相乘，存高位

说明：源和目的操作数的 4 个带符号字分别相乘，并把结果的高 16 位存入目的寄存器相应字中。实现 MMX 寄存器之间、内存到 MMX 寄存器之间的操作。

示例：PMULHW XMM3,XMM1

PMULHW XMM3,MEM_QUAD

25. PMULLW 紧缩带符号字相乘，存低位

说明：参考第 24 指令——PMULHW

示例：PMULLW XMM4,XMM7

PMULLW XMM4,MEM_QUAD

26. POR 逻辑或

说明：对应位逻辑或。实现 MMX 寄存器之间、内存到 MMX 寄存器之间的操作。

示例：POR XMM1,XMM3

POR XMM1,MEM_QUAD

27. PSLLD 紧缩双字逻辑左移

说明：把 MMX 寄存器的 2 个双字分别作逻辑左移。目标操作数一定是 MMX 寄存器，移位位数可放在 MMX 寄存器中、内存中或立即数。

示例：PSLLDXMM0,XMM3

PSLLD XMM0,3

PSLLD XMM0,MEM_QUAD

28. PSLLQ 紧缩 4 字逻辑左移

说明：请参考第 27 条指令——PSLLD。

示例：PSLLQXMM0,XMM3

PSLLQ XMM0,3

PSLLQ XMM0,MEM_QUAD

29. PSLLW 紧缩字逻辑左移

说明：请参考第 27 条指令——PSLLD。

示例：PSLLW XMM0,XMM3
PSLLW XMM0,3
PSLLW XMM0,MEM_QUAD

30. PSRAD 紧缩双字算术右移

说明：把 MMX 寄存器的 2 个双字分别作算术右移，高位补符号。目标操作数一定是 MMX 寄存器，移位位数可放在 MMX 寄存器中、内存中或立即数。

示例：PSRAD XMM6,XMM1
PSRAD XMM6,MEM_QUAD
PSRAD XMM6,200

31. PSRAW 紧缩字算术右移

说明：请参考第 30 条指令——PSRAD。

示例：PSRAW XMM6,XMM1
PSRAW XMM6,MEM_QUAD
PSRAW XMM6,16

32. PSRLD 紧缩双字逻辑右移

说明：把 MMX 寄存器的 2 个双字分别作逻辑右移，高位补 0。目标操作数一定是 MMX 寄存器，移位位数可放在 MMX 寄存器中、内存中或立即数。

示例：PSRLD XMM4,XMM2
PSRLDXMM4,MEM_QUAD
PSRLDXMM4,123

33. PSRLQ 紧缩 4 字逻辑右移

说明：请参考第 32 条指令——PSRLD。

示例：PSRLQ XMM4,XMM2
PSRLQXMM4,MEM_QUAD
PSRLQXMM4,123

34. PSRLW 紧缩字逻辑右移

说明：请参考第 32 条指令——PSRLD。

示例：PSRLW XMM4,XMM2
PSRLW XMM4,MEM_QUAD
PSRLW XMM4,123

35. PSUBB 紧缩字节减法，溢出则回绕

说明：把目标操作数的 8 个字节减去对应源操作数的 8 字节。可以是 MMX 寄存器

减 MMX 寄存器或 MMX 寄存器减内存的内容。

结果溢出则回绕。关于回绕，请参考第 7 条指令——PADDB。

示例：PSUBB XMM6,XMM2

PSUBB XMM5,MEM_QUAD

36. PSUBD 紧缩双字减法，溢出则回绕

说明：请参考第 35 条指令——PSUBB。

示例：PSUBD XMM2,XMM2

37. PSUBW 紧缩字减法，溢出则回绕

说明：请参考第 35 条指令——PSUBB。

示例：PSUBW XMM6,XMM4

38. PSUBSB 带符号紧缩字节减法，溢出则取饱和值。

说明：把目标操作数的 8 个带符号字节减去对应源操作数的 8 个带符号字节。可以是 MMX 寄存器减 MMX 寄存器或 MMX 寄存器减内存的内容。

结果溢出则取饱和值。关于饱和值，请参考第 4 条指令——PACKSSDW。

示例：PSUBSB XMM5,XMM2

PSUBSB XMM3,MEM_QUAD

39. PSUBSW 带符号紧缩字减法，溢出则取饱和值。

说明：请参考第 38 条指令——PSUBSB。

示例：PSUBSW XMM5,XMM2

40. PSUBUSB 无符号紧缩字节减法，溢出则取饱和值。

说明：把目标操作数的 8 个无符号字节减去对应源操作数的 8 个无符号字节。可以是 MMX 寄存器减 MMX 寄存器或 MMX 寄存器减内存的内容。

结果溢出则取饱和值。关于饱和值，请参考第 4 条指令——PACKSSDW。

示例：PSUBUSB XMM2,XMM7

PSUBUSB XMM2,MEM_QUAD

41. PSUBUSW 无符号紧缩字减法，溢出则取饱和值。

说明：请参考第 40 条指令——PSUBUSB。

示例：PSUBUSW XMM2,XMM7

42. PUNPCKHBW 反紧缩高位——字节到字

说明：把源操作数和目标操作数反紧缩并把高位交替放入目标操作数中，如图 A-4

所示。实现 MMX 寄存器之间、内存到 MMX 寄存器的操作。

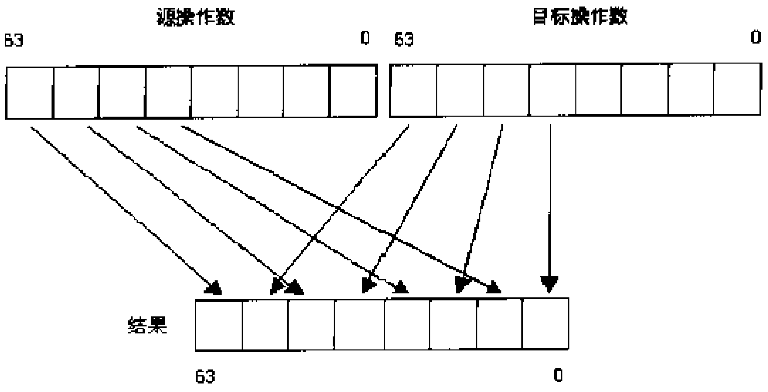


图 A-4

示例：PUNPCKHBW XMM4,XMM6

PUNPCKHBW XMM3,MEM_QUAD

43. PUNPCKHDQ 反紧缩高位——双字到 4 字

说明：请参考第 42 条指令——PUNPCKHBW。

示例：PUNPCKHDQ XMM4,XMM6

44. PUNPCKHWD 反紧缩高位——字到双字

说明：请参考第 42 条指令——PUNPCKHBW。

示例：PUNPCKHWD XMM4,XMM6

45. PUNPCKLBW 反紧缩低位——字节到字

说明：把源操作数和目标操作数反紧缩并把低位交替放入目标操作数中。实现 MMX 寄存器之间、内存到 MMX 寄存器的操作。

说明：请参考第 42 条指令——PUNPCKHBW。

示例：PUNPCKLBW XMM4,XMM6

PUNPCKLBW XMM3,MEM_QUAD

46. PUNPCKLDQ 反紧缩低位——双字到 4 字

说明：请参考第 45 条指令——PUNPCKLBW。

示例：PUNPCKLDQ XMM4,XMM6

47. PUNPCKLWD 反紧缩低位——字到双字

说明：请参考第 45 条指令——PUNPCKLBW。

示例：PUNPCKLWD XMM4,XMM6

48. PXOR 逻辑异或

说明：源和目标操作数按位逻辑异或。实现 MMX 寄存器之间和内存到 MMX 寄存器的操作。

示例：PXOR XMM7,XMM3

PXOR XMM3,MEM_QUAD

因为 TASM 5.0 不支持 MMX 指令，所以如果要在 TASM 5.0 中使用 MMX 指令，则要自己定义宏。这是一件比较麻烦的事，因为用户须找出各指令对应的操作码。

在前面中提到的 MAXM 32 可以直接使用 MMX 指令。只要在源文件的开头加上“.XMM”即可以在文件的下面使用 MMX 指令。

下面的源程序 MMX.ASM 测试上面介绍的全部 MMX 指令。

```
;*****************************
;文件：MMX.asm              *
;功能：测试 MMX 指令         *
;*****************************
    .686P
    .xmm

    .model flat, stdcall
    option casemap :none

    include d:\masm32\include\windows.inc        ;头文件
    include d:\masm32\include\user32.inc
    include d:\masm32\include\kernel32.inc
    includelib d:\masm32\lib\kernel32.lib    ;库文件
    includelib d:\masm32\lib\user32.lib

    .DATA                              ;数据
    Caption        db '测试 MMX 指令',0
    Text_Begin     db '开始',0
    Text_End       db '成功结束',0

    align 2
    MEM_BYTE  DB  ?
    align 2
    MEM_WORD DW  ?
    align 4
    MEM_DWORD      DD  ?
    align 8
    MEM_QUAD       QWORD  ?
```

```
;*****************************************************************
      .CODE                          ;代码
Start:
      Invoke MessageBox,NULL,offset Text_Begin,offset Caption,MB_OK

      ;----------------------
      EMMS

      ;----------------------
      MOVD    XMM0 ,MEM_DWORD
      MOVD    MEM_DWORD,XMM1
      MOVD    XMM2,EAX
      MOVD    EBX,XMM3

      ;----------------------
      MOVQ    XMM4,XMM5
      MOVQ    XMM5,MEM_QUAD

      ;----------------------
      PACKSSDW XMM0,XMM1
      PACKSSDW XMM0,MEM_QUAD

      ;----------------------
      PACKSSWB XMM0,XMM1
      PACKSSWB XMM0,MEM_QUAD

      ;----------------------
      PACKUSWB XMM0,XMM1
      PACKUSWB XMM0,MEM_QUAD

      ;----------------------
      PADDB  XMM6,XMM7
      PADDB  XMM0,MEM_QUAD
      PADDW  XMM6,XMM7
      PADDD  XMM6,XMM7

      ;----------------------
      PADDSB XMM1,XMM2
```

```
PADDSB XMM0,MEM_QUAD

PADDSW      XMM1,XMM2

;-----------------------
PADDUSB     XMM0,XMM1
PADDUSB     XMM0,MEM_QUAD
PADDUSW     XMM0,XMM1

;-----------------------
PAND    XMM4,XMM1
PAND    XMM4,MEM_QUAD

;-----------------------
PANDN  XMM4,XMM1
PANDN  XMM4,MEM_QUAD

;-----------------------
PCMPEQB     XMM0,XMM1
PCMPEQB     XMM0,MEM_QUAD

PCMPEQW    XMM0,XMM1

PCMPEQD    XMM0,XMM1

;-----------------------
PCMPGTB     XMM3,XMM1
PCMPGTB     XMM2,MEM_QUAD

PCMPGTW    XMM3,XMM1

PCMPGTD     XMM3,XMM1
;-----------------------
PMADDWD   XMM5,XMM2
PMADDWD   XMM6,MEM_QUAD
;-----------------------
PMULHUW XMM3,XMM1
PMULHUW XMM3,MEM_QUAD
```

```
PMULHW XMM3,XMM1
PMULHW XMM3,MEM_QUAD
;----------------------
PMULLW XMM4,XMM7
PMULLW XMM4,MEM_QUAD
;----------------------
POR     XMM1,XMM3
POR     XMM1,MEM_QUAD
;----------------------
PSLLW   XMM0,XMM3
PSLLW   XMM0,3
PSLLW   XMM0,MEM_QUAD
PSLLD   XMM0,XMM3
PSLLD   XMM0,3
PSLLD   XMM0,MEM_QUAD
PSLLQ   XMM0,XMM3
PSLLQ   XMM0,3
PSLLQ   XMM0,MEM_QUAD
;----------------------
PSRAW   XMM6,XMM1
PSRAW   XMM6,MEM_QUAD
PSRAW   XMM6,16
PSRAD   XMM6,XMM1
PSRAD   XMM6,MEM_QUAD
PSRAD   XMM6,200
;----------------------
PSRLW   XMM4,XMM2
PSRLW   XMM4,MEM_QUAD
PSRLW   XMM4,123
PSRLD   XMM4,XMM2
PSRLD   XMM4,MEM_QUAD
PSRLD   XMM4,123
PSRLQ   XMM4,XMM2
PSRLQ   XMM4,MEM_QUAD
PSRLQ   XMM4,123
;----------------------
PSUBB   XMM6,XMM2
PSUBB   XMM5,MEM_QUAD
```

```
    PSUBW  XMM6,XMM4
    PSUBD  XMM2,XMM2
    ;-----------------------
    PSUBSB XMM5,XMM2
    PSUBSB XMM3,MEM_QUAD
    PSUBSWXMM5,XMM2
    ;-----------------------
    PSUBUSB    XMM2,XMM7
    PSUBUSB    XMM2,MEM_QUAD
    PSUBUSW    XMM2,XMM7
    ;-----------------------
    PUNPCKHBW XMM4,XMM6
    PUNPCKHBW XMM3,MEM_QUAD
    PUNPCKHWD XMM4,XMM6
    PUNPCKHDQ XMM4,XMM6
    ;-----------------------
    PUNPCKLBW XMM4,XMM6
    PUNPCKLBW XMM3,MEM_QUAD
    PUNPCKLWD XMM4,XMM6
    PUNPCKLDQ XMM4,XMM6
    ;-----------------------
    PXOR    XMM7,XMM3
    PXOR    XMM3,MEM_QUAD
    ;-----------------------
    Invoke MessageBox,NULL,offset Text_End,offset Caption,MB_OK
    Invoke ExitProcess,0
;*************************************************************
    End Start
```

用下面的命令编译这个程序：

```
F:\mbk\nbp\Appendix\MMX>D:\masm32\bin\ml -c -coff MMX.asm
Microsoft (R) Macro Assembler Version 6.14.8444
Copyright (C) Microsoft Corp 1981-1997.  All rights reserved.
Assembling: MMX.asm
F:\mbk\nbp\Appendix\MMX>D:\masm32\bin\link -subsystem:windows mmx.obj
Microsoft (R) Incremental Linker Version 5.12.8078
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.
```